DATA MANAGEMENT IN DYNAMIC DISTRIBUTED COMPUTING ENVIRONMENTS

BY

IAN ROBERT KELLEY

A dissertation submitted to the faculty of

Cardiff University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Cardiff School of Computer Science & Informatics

Cardiff University

June 2012

Acknowledgements

The journey leading to this dissertation's completion has been long, with many challenges and unexpected twists and turns along the way. It all began nearly nine years ago, when the idea to pursue a computer science doctorate was conceived during my time as a research programmer in Berlin. This notion evolved and became a goal of mine, which I pursued during my subsequent move to Louisiana. Two years later, I was accepted into a Ph.D. program at Cardiff University, where I began my research in the field of distributed systems.

Throughout this experience, I have made many new friends and colleagues, and I thoroughly enjoyed life at Cardiff University, first as a student and later as a researcher. I would like to thank many of the people I encountered on the road leading here, not only for their help in my research, but most of all for their friendship. First, I would like to recognize Edward Seidel and Gabrielle Allen, who were there in the very beginning and started me on this path. Next, and of special note, are the members of "the group" at Cardiff: Matthew Shields, Andrew Harrison, and my supervisor (and friend) Ian Taylor. Each of you, in your own way, made my time in the U.K. more enjoyable and fruitful. I miss our lively after-work debates and late-night curries.

I am grateful to the Cardiff School of Computer Science & Informatics, both for the support its members have shown me and also for making this all possible. I would like to acknowledge my colleagues from the Enabling Desktop Grids for e-Science and European Desktop Grid Initiative projects, who contributed many ideas and helped drive my research.

Lastly, I appreciate the support given to me over the years by my family and friends. You have constantly reminded me to complete my dissertation, which finally resulted in pen being put to paper (or, rather, fingers to keys) at long last. I would especially like to thank my aunt Victoria Scott and my grandmother, Carolyn Scott, for their exceptional editorial assistance. My grandmother was not only steadfast in her encouragement, but also came out of retirement at age 90 to proofread my dissertation!

I dedicate this work to my son Nils and my partner Erica Hammer. At the time of this writing Nils is 11, yet was much younger when it all began. I hope one day he will appreciate the sacrifices that were made so this research could see the light of day.

Abstract

Data management in parallel computing systems is a broad and increasingly important research topic. As network speeds have surged, so too has the movement to transition storage and computation loads to wide-area network resources. The Grid, the Cloud, and Desktop Grids all represent different aspects of this movement towards highly-scalable, distributed, and utility computing.

This dissertation contends that a peer-to-peer (P2P) networking paradigm is a natural match for data sharing within and between these heterogeneous network architectures. Peer-to-peer methods such as dynamic discovery, fault-tolerance, scalability, and ad-hoc security infrastructures provide excellent mappings for many of the requirements in today's distributed computing environment.

In recent years, volunteer Desktop Grids have seen a growth in data throughput as application areas expand and new problem sets emerge. These increasing data needs require storage networks that can scale to meet future demand while also facilitating expansion into new data-intensive research areas. Current practices are to mirror data from centralized locations, a technique that is not practical for growing data sets, dynamic projects, or data-intensive applications.

The fusion of Desktop and Service Grids provides an ideal use-case to research peer-to-peer data distribution strategies in a hybrid environment. Desktop Grids have a data management gap, while integration with Service Grids raises new challenges with regard to cross-platform design. The work undertaken here is two-fold: first it explores how P2P techniques can be leveraged to meet the data management needs of Desktop Grids, and second, it shows how the same distribution paradigm can provide migration paths for Service Grid data.

The result of this research is a Peer-to-Peer Architecture for Data-Intensive Cycle Sharing (ADICS) that is capable not only of distributing volunteer computing data, but also of providing a transitional platform and storage space for migrating Service Grid jobs to Desktop Grid environments.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

Data management in distributed and heterogeneous computing environments can be complex. In the parallel computing domain, there exist a number of solutions for the movement and storage of files, often tailored to specific architectures or optimized for a particular set of applications. As network speeds have increased, so too has the movement to transition storage and computation loads to wide-area network resources, to facilitate service interconnection, scalability, and utility computing.

Migrating applications to distributed resources presents a number of challenges, among them increased data loads and their subsequent management. Centralized or static solutions that were efficient in tightly-coupled local area network (LAN) configurations can quickly become unwieldy bottlenecks in the new environment, either due to an inability to scale or otherwise adapt to changing network dynamics. One computing paradigm that is currently confronting these data challenges is Desktop Grids, notably within the highly successful volunteer computing domain.

Volunteer computing has become a popular and successful means of providing vast amounts of processing power to scientists for little or no direct cost. This is achieved through the creation of a publicly open system where private individuals (i.e., “volunteers”) install a software program that takes advantage of their computer's processing power when it would otherwise be idle. This gathering and harnessing of volunteer computing resources has enabled scientific projects to aggregate hundreds of thousands of idle processors for their research, far beyond anything even the world's largest and most expensive supercomputers could have provided.

The SETI@Home[1] project was instrumental in both the popularity of volunteer computing and the technical achievements that made it possible. Launched to the public in 1999, two years after Distributed.net broke new ground, it became the second large-scale research project to use distributed computing over the Internet. SETI@Home was successful in popularizing volunteer computing, and was highly publicized, with a staggering 300,000 computers donating time to the project within just one week of its launch.

In a volunteer computing scenario, such as that created by SETI@Home, distributed network resources "donate" their idle cycles to a particular project on a voluntary basis. By aggregating the vast computing resources of these volunteer nodes, scientific projects are able to perform massive amounts of calculations for a relatively low cost that is generally limited to the administrative overhead of managing the network. When SETI@Home was released to the public, it enabled tens of thousands of non-scientists to contribute to science by installing a simple "screen saver" on their home computers. Behind the scenes, the program was in reality doing much more than preventing screen burn-in: it was analyzing vast amounts of radio telescope data, all part of SETI's search for extraterrestrial intelligence.

It has been over a decade since SETI@Home first started using personal computers to analyze data, and although in the vastness of space we have yet to find extraterrestrial intelligence, the project has had a significant effect in the broader scientific community. Three years after SETI@Home's launch, the Berkeley Infrastructure for Open Network Computing (BOINC) was released to the public. BOINC was the result of a rewrite of the original SETI@Home code-base, but was written to be generic for any scientific application that needed to process large numbers of independent simulations. BOINC provided a way for scientists to "farm out" their parallel processing jobs to computers distributed around the globe and opened the door for numerous other scientific projects to join the volunteer computing movement.

[1]SETI@Home grew out of the needed to analyze large amounts of radio-telescope data being produced by the Search for Extraterrestrial Intelligence (SETI). For more information, see §2.3.

Figure 1.1: Arecibo Radio Telescope — *The 1000 ft. (305 m) diameter telescope, located in Puerto Rico, is the largest single-aperture telescope ever constructed. It collects radio waves that are analyzed by the SETI@Home software. Some might recognize it as the setting for the end-scenes in the James Bond movie "Goldeneye," as well as a good portion of the movie "Contact."*

With BOINC leading the way, volunteer computing became a very successful way to harness the hardware resources of personal computers – to date laying claim to more than 10,000 petaflop/days of computation.[2] The applications that are able to make use of this revolutionary computing paradigm are diverse, with over 50 projects using BOINC as their main processing engine, and scientific application domains ranging from genome sequencing to Pi calculations. This is a very impressive achievement, and it has been instrumental in the success of many projects that would otherwise have had to vie for much more limited computational resources.

To date, with little exception, BOINC and other volunteer computing environments like it have focused on exploiting idle Central Processing Unit (CPU) cycles – the lowest hanging fruit, with the least impact on donated resources. More recently, efforts have been made to leverage the power of fast Graphical Processing Units (GPUs), which can many times more powerful than consumer CPUs in floating point calculations [1]. With easily available CPU and GPU resources, focus has been on supporting highly parallel (or even embarrassingly parallel) processor-intensive applications that can take full advantage of a machine's number-crunching abilities. However, other available resources, such as

[2]As of this writing, the total BOINC credit is over 1 trillion "Cobblestone." Each 100 "Cobblestone" represents a day of sustained computation at a gigaflop.

hard-disk storage capacities and high-speed network connections, have yet to be tapped for their potential.

With most home computers now equipped with large underutilized hard drives and sharp increases in the speed of personal Internet connections, it has recently become both feasible and attractive to extend these volunteer networks beyond CPU-harvesting to include the distribution and storage of scientific content. Doing so would increase the range of applications that can run on these highly distributed systems to include those that have larger data-processing requirements. Utilizing volunteer networks and storage can greatly reduce the data scalability bottlenecks that can quickly occur when distributing even moderately sized data-sets to tens of thousands of participants. The problem that occurs is one of simple scalability. Applications that currently use volunteer computing generally have very low data throughput demands. For example, a work-unit for SETI@Home is only a few hundred kilobytes, and those of Einstein@Home are only several megabytes. Both can run for several hours, meaning that a processing node will only be requesting a few megabytes each day at most. Multiply that by the tens of thousands of clients subscribed to a project and the data needs become very large and potentially expensive, yet perhaps still manageable for a large, organized, and successful project that is able to increase its data replicas as its user base expands. Contrast this current scenario to a bioinformatics or image-rendering application, where a node might need hundreds of megabytes, or potentially gigabytes, for a single simulation. The data distribution needs in such an application domain can easily explode far beyond any reasonable ability to maintain project-level servers and mirrors. The problem compounds as the network scales up, and can eventually render the application unsuitable for a highly distributed environment.

Current scientific volunteer computing software infrastructures such as BOINC and XtremWeb, which are discussed in §2.3, distribute their data centrally from a project's coordinating nodes or servers. In BOINC, this is achieved through a set of HTTP mirrors, each providing clients with full copies of the data input files needed for a particular simulation. Similarly, in XtremWeb clients are given the URIs of data input files, which, although not as coordinated as BOINC, is still a centralized distribution model. These systems require projects not only to have the necessary network capacity needed to provide data to all volunteers, but also to have data readily available and persistent on their servers at all times to fulfill client requests. Given that data distribution is still a direct cost incurred by a project, the network throughput requirements of serving so many client machines can prove to be a hinderance when needing to explore new types of data-intensive application scenarios, ones that are currently prohibitive in terms of their large data-transfer needs. To maintain data scalability, most current scientific projects that utilize volunteer computing scale down their problem sizes and/or data resolutions to fit into the bandwidth and storage capacities they have available. Similarly, projects that have large data needs shy away from using volunteer computing, knowing that their data requirements could quickly exceed their hosting abilities.

A viable alternative to such limiting centralized systems is to employ the use of peer-to-peer (P2P) techniques to implement data distribution. “Peer-to-peer” is used to describe a system where the consumers of information or data are also the providers. Unlike a traditional web-server, where data are served by one or more centralized locations and consumed by many, in a P2P system data are relatively evenly distributed among the network participants, with roles being interchangeable depending on the needed network topology and requirements at any given time.

Volunteer computing is a complementary ideology to the P2P computing paradigm, in which “many hands make light work” and responsibility is shared among a large set of participants. “Volunteer computing” itself is more of a concept than a particular technological implementation. Very often volunteer networks are referred to as “Desktop Grids” (DGs) due to the general notion that they are comprised of desktop-grade consumer-oriented computers. One can easily argue that the processing in BOINC is very peer-to-peer in the ideological sense with respect to processing power burden-sharing. However, job distribution, aggregation, and data management remain very centralized. The BOINC software manages the network connections and data distribution in a strict master/worker relationship, with all data being transferred solely between an organization’s central servers and volunteer nodes. There is no data-sharing or information exchange between individual network participants.

By using a P2P network to distribute scientific data, the individual participants (i.e., volunteer resources) could help to share input data with one another, changing the current master/worker paradigm. This is useful because due to the nature of volunteer computing networks and the inherent calculation errors that occur, the same simulation is often sent to more than one node for processing, with the results then being compared to validate any given response. Although effective in garnishing correct calculations, this requires that all data be distributed twice, or in some cases three times, for each simulation. Repeat sending of data is compounded even more when the same data are used for multiple simulations, such as in Monte Carlo simulations or parameter sweep applications. The most obvious case of data redundancy would be for large shared data-sets that need to be transferred to each individual network node, such as a large bioinformatics database, thereby requiring the centralized servers to send the same set of data to N participants, where N is the size of the entire network. Even with a relatively small volunteer pool of 10,000 nodes, a 1-gigabyte input file can become a painful 10-terabyte distribution problem. Peer-to-peer data distribution could help offload these requirements to the volunteer network, potentially reducing the centralized data requirements to one distribution per file, a reduction of many orders of magnitude.

Another very large potential scientific user-base for Desktop Grids that has yet to be explored is that of the individual researcher. BOINC has been successful in providing large, visible, and popular computing projects with additional computational resources; however, smaller, less visible groups and individual researchers have been largely left out. This is largely due to the infrastructure, both technological and marketing, that is required to have a successful BOINC project. Beyond defining the scientific problem in a way that can be processed in parallel as a "bag of tasks," potential users must deploy servers to host input files, keep track of databases, and, the most difficult task of all, recruit willing volunteers to donate their resources towards a particular cause.

This can work well for sufficiently capable groups that have long-term simulation needs, where a high initial investment will easily pay off through years of sustained "free" computation. However, the setup and maintenance costs are impractical, if not impossible, for individual researchers, Ph.D. students, and small research groups, whose computation needs may be transitory. For example, a Ph.D. student might need 1,000 CPU years to complete an analysis for his or her dissertation, or a researcher might want to simulate the results of a small grant, after which the infrastructure would be dormant until the next need arose. For these types of users, university labs, Condor pools, and national computing centers (i.e., clusters and supercomputers), otherwise known as "Service Grids" (SGs), are a more hospitable environment.

As a result, in many cases, large numbers of highly parallel (i.e., "embarrassingly parallel") simulations are being run on expensive Service Grid infrastructures, namely, supercomputers and clusters with complex storage area networks

(SANs), large memory allocations, and high-speed network interconnects that are most cost-effective for running the heavy processor-interdependent MPI applications for which they were designed. This is a very inefficient use of Service Grid resources for simulations that could otherwise be run on a Desktop Grid. One of the challenges of moving these applications to a volunteer network or other Desktop Grid system is how to distribute the data. On clusters and supercomputers, data are generally located in a secured storage management system and must be made available at some publicly addressable space before they can be downloaded by volunteer clients. Even if it were possible to host a common BOINC server for these researchers, to process results and distribute relatively small work-units and parameter files, the data distribution demands could easily overwhelm any centralized architecture. This is especially true if the goal is to offload the widest range of parallel applications, including those with larger input files. A peer-to-peer system for data distribution could provide a means to decentralize the data distribution, as well as provide a data bridge between supercomputer/-cluster and volunteer environments.

Not only would application of a dynamic data-distribution network reduce the Service Grid resources needed to integrate with or migrate jobs to a Desktop Grid, it could also mitigate the potential risk involved when moving jobs and data to the Desktop Grid. By providing an intermediary layer, one is able to limit the number of peers to which a Service Grid node must distribute data. This can be further refined by applying project-based security criteria to govern the membership composition of the data brokers. For the Desktop Grid network, a P2P data distribution system would also allow current projects to take full advantage of client-side network and storage capabilities, enabling the exploration of new types of data-intensive application scenarios, ones that are currently overly prohibitive given their large data transfer needs.

There are many ways peer-to-peer data distribution, or a variant thereof, could be achieved for volunteer networks, ranging from BitTorrent-style distribution, where data are centrally tracked and all participants share relatively equal loads, to KaZaa-like super-peer networks, where select nodes are assigned greater responsibility in the network and broadcast messages and network flooding are used to locate information. However, applying a traditional P2P network infrastructure to scientific computing, and in particular to volunteer computing, can be highly problematic. In such environments, policies and safeguards for scientific data and users' computers are critical concerns that limit uptake, rather than the technical feasibility of developing a solution. Identifying and adapting to the environment and its constraints, discussed further in chapters 2 and 3, is a crucial aspect in applying a new technology in the volunteer computing domain.

A tailor-made solution that could take into account the requirements of scientific communities, as opposed to a generic overarching P2P architecture, would have the advantage of facilitating different network topologies and data distribution algorithms while retaining the safety of each participant's computer. Further, each scientific application has different network and data needs, and customized solutions would allow for tailoring the network towards individual requirements, albeit with the disadvantage of increased development effort, complexity, and code maintenance. Any P2P system would have to take into consideration the specific needs of scientific volunteer computing applications, such as control of security aspects, legacy application integration, and mechanisms for participants to opt-in and opt-out of the system.

In the research presented here, I work to show how volunteer computing's narrow scope of CPU and GPU harvesting could be expanded to also include the utilization of network and storage capabilities. Specifically, I will justify pursuing a P2P-based approach for data distribution and demonstrate how decentralized P2P networks can be built to distribute scientific data successfully. As I explore the applicability of a peer-to-peer data-sharing paradigm to volunteer computing networks, I investigate how P2P technologies and client-side file sharing need to be adapted to suit the needs of a volunteer-driven computing pool, and what this means for both the users and the technology involved.

1.1 Thesis Goals

A hypothesis is a proposed explanation for a phenomenon, or a statement of supposition to be used as the premise for an argument. For a scientific hypothesis to be validated, one must be able to test and probe it, and subject it to the scientific method. After these trials, the hypothesis should be accepted and considered as valid, or disregarded as false or unprovable.

This work is the thesis to provide support for the following hypothesis:

Peer-to-peer file sharing can be successfully applied to the volunteer computing application domain. Through the use of distributed and cached data on *dynamic networks, both large- and small-scale scientific projects will be able to more effectively scale in a reliable and secure manner as their projects expand.*

The aforementioned hypothesis was proposed as a way to fill a need within the volunteer computing community for a robust data-management solution that can scale as network participation and demand increases. The goal of this thesis is to provide evidence and validation for the hypothesis by investigating the feasibility and usefulness of applying a peer-to-peer data distribution paradigm to volunteer networks. The proof of the hypothesis depends not only on the presentation of a technological solution that "can" distribute data, but also on whether or not the solution is acceptable to the community and can therefore be considered useful, valid, and novel.

The additional and narrower goals of this thesis are to show how P2P technologies can provide a low-cost and low-maintenance alternative to the master/-worker data-distribution approach currently used in Desktop Grids. One of the promises of peer-to-peer computing is to leverage the use of spare cycles and storage in personal computers, thus providing an infrastructure that can perform complex scientific tasks. An important issue in building P2P systems is providing an easy way to access, manage, and use these volatile distributed computational resources, lowering the cost of adaptation and making P2P networks an attractive option for scientific data distribution. The key element to the research is showing how P2P provides an extremely well-matched and suitable, perhaps even best-fit, tool for distributing data in volunteer computing network environments.

BOINC-type projects, which traditionally rely on multiple iterations of calculations over the same data chunk, provide an ideal use-case for a scientific application that would benefit from P2P. Once the data enter the network, they could potentially be distributed in a decentralized manner among individual peers. This would allow the centralized servers to inject the data into the network a limited number of times (ideally one) and therefore require only a fraction of the currently used total overall bandwidth to distribute the data to all interested parties. By utilizing a P2P data distribution approach, projects should be able to reduce their network overhead, not only allowing for cost-cutting, but also facilitating the distribution and analysis of higher-resolution data-sets that are currently being limited by available network capabilities.

Advanced P2P environments have been built in the past. However, most are geared toward solving a particular problem and do not provide a generic solution that can be plugged into any application layer. Many examples exist, particularly in the realm of commercial data distribution, for example, KaZaa, Napster, Gnuttella, and Skype. However, the lack of generic, open, and standard P2P tools and infrastructures has often made the additional cost of adopting a P2P environment outweigh the potential benefits for scientific application groups. This has severely limited its integration and uptake.

It is with respect to the more sophisticated and narrow requirements of volunteer computing, and Desktop Grids in general, that I propose and present here a customizable and brokered Peer-to-Peer Architecture for Data-Intensive Cycle Sharing (ADICS) that allows fine-grained provisioning of resources and application of project-based roles to network participants. Specifically, ADICS provides a brokered P2P system that offloads central network needs while limiting client exposure to foreign hosts, thus providing a viable alternative to centralized data distribution. The brokered network-overlay introduced in ADICS acts as a buffer, in the form of a select group of trusted data-sharing nodes, between the central BOINC server, or even a Service Grid, and the Desktop Grid data consumers. Moreover, through the adaptation of P2P data distribution techniques, centralized network resource requirements are significantly reduced while fault tolerance and accessibility are increased.

Beyond arguing the theoretical usefulness of a dynamic distribution paradigm for volunteer computing, I present a P2P-based solution that conforms to the requirements of a volunteer computing environment. Additionally, application developers are provided with straightforward migration paths to transition from traditional client/server models to ADICS. Throughout the work presented here, particular emphasis is given to, and research focused on, the key areas of P2P overlay networks, security, scalability, and data integrity.

Specifically, the goals of this thesis are to verify the hypothesis by:

- Conceptualizing how a new data-distribution method for volunteer computing could be realized.

  Is Peer-to-peer a suitable and efficient data-distribution technique for scientific volunteer computing environments in which data have traditionally been exchanged in a centralized one-to-many relationship?

- Identifying whether there are scientific projects that currently find transition to volunteer computing platforms prohibitive due to data challenges.

- Researching security and scalability issues, while identifying potential tradeoffs and defining optimal solutions.

- Isolating useful and novel data-sharing mechanisms that might be adaptable to the target environment, while maintaining network security parameters and user trust.

Research Issues

During the course of this thesis, through the fusion of the goals mentioned in §1.1 and the investigations performed in chapters 2 and 3, the following basic principles have been identified as within the scope of the thesis and applicable to the solution and application domain:

- Solutions must ensure that a high level of security and trust be retained in the network, while maintaining the integrity of data and preventing unauthorized usage.
- Transparent access from existing distributed processing applications, specifically BOINC-based scientific projects, such as Einstein@Home, must be provided.
- Network design must be able to cope with the large size of scientific computing projects.

While upholding the aforementioned principles, the following questions arise:

- How can the architecture of a data-sharing framework be developed so that it can be used generically to distribute large datasets?
- What are the hooks for target applications that will enable them to take advantage of P2P data-sharing networks?
- Can Peer-to-peer technologies be applied to scientific applications in secure and beneficial ways, without additional cost or substantial risk?
- Should aspects of newly adopted solutions such as Grid technology be combined with peer-to-peer to enable new scenarios?

  e.g., X.509 certificates, decentralized data stores, resource brokering

- Can generic peer-to-peer network building and negotiation be furthered through use of standardized mechanisms such as XML?

  What is the tradeoff between XML overhead and enhanced interoperability and description capabilities?

Evaluation

Given the goals of this thesis, it is expected that the evaluation may include the following:

- Research supports the hypothesis (security evaluations, scalability trials and simulations) or does not.
- Comparison of the thesis's data distribution proposals with the current static and centralized systems.
- The concept can be proven to work, either in "real life" or in a "network simulator."
- Thesis ideas and resulting software are used by other researchers or end-users, either directly or as the basis for further work.

1.2 Thesis Overview

In this thesis, I outline the reasons for, the challenges in, the benefits of, and the progress towards applying a Peer-to-Peer Architecture for Data-Intensive Cycle Sharing (ADICS).

The work presented here is divided into seven chapters: Chapter 1, which you are currently reading, introduces the subject matter; Chapter 2 delves into the background, related work, and motivation; Chapter 3 analyzes the issue and discusses the methodology; Chapter 4 proposes a new network architecture; Chapter 5 gives the implementation details and shows the broader impact; Chapter 6 provides a critical assessment and discusses further work; and, lastly, Chapter 7 concludes.

Pursuant to the research goals that I have identified for the thesis, the rest of this work is organized as follows:

Chapter 2: Background, Motivation, and Related Work

This chapter gives background on the technologies currently employed for scientific computing, the environment and the problem scope of volunteer computing and its associated data. Related work is discussed and use-cases are given for how a data management solution is needed, highlighting the technological gaps that provide the basis and motivation for the research undertaken here.

Chapter 3: Analysis and Design

Perhaps the most innovative chapter, Chapter 3 delves into an analysis of the problem, its requirements, and the limitations that must be overcome. The "plan of attack" and methodology that will be used throughout the research are presented, showing how these relate to the goals of applying a distributed data management solution to Desktop Grids.

Chapter 4: Peer-to-Peer Architecture for Data-Intensive Cycle Sharing

Here, a Peer-to-Peer Architecture for Data-Intensive Cycle Sharing (ADICS) is introduced. ADICS serves as the data distribution paradigm and architectural model for the Desktop Grid data distribution presented in this thesis. Building upon Chapter 3, this chapter relates the proposed system to the requirements of volunteer computing, and shows how it is able to provide a new and novel way of distributing input data.

The chapter first shows the general architecture of ADICS, comparing it against the requirements of the target community. Next, it gives the simulation and mathematical results that are used to help support the thesis and lead design. The basic design principles are referenced to the use-cases introduced in Chapter 2, and the major user requirements are outlined. In addition, architectural strategies that were employed during the design of the overall system are discussed, as well as the details of the system and its entities.

Chapter 5: Implementation and Integration with Service and Desktop Grids

Chapter 5 gives the implementation details of the software that was designed to support the thesis. The tactics and guidelines that were used in the software design process are explained. Additionally, details are supplied of how the architected system provides enhancements and support for two large European Union infrastructure projects, applying the thesis principles to real-world use-cases and integration issues.

Chapter 6: Assessment and Further Work

Chapter 6 shows the impacts of the research, and how it has been applied and used beyond original expectations to produce new results and innovation. Critical assessment of the approach proposed by the thesis and the software developed to support it are examined, identifying both the strengths and weaknesses. The chapter also suggests how the thesis and the software could be improved and enhanced to serve research communities better, and thus expand the potential impact of the approach presented here.

Chapter 7: Conclusions

The last chapter provides a synopsis of the thesis, the work that has resulted from it, and the novelty of the approach and its impact. As a final contribution, the chapter also suggests additional research topics for how the work presented in this thesis could be expanded into broader application domains.

Appendices

In addition to the main text, there are three appendices. **Appendix A** gives code examples and logistical instructions for the software developed in this project. **Appendix B** gives details of the projects that have furthered the thesis software and goals. Lastly, **Appendix C** gives a list of my publications that are relevant to the research presented in this dissertation.

1.3 Contributions

The focus of this thesis is on exploring how peer-to-peer data distribution could work to satisfy the data needs of a volunteer computing environment. The architecture being proposed here (i.e., ADICS) is a decentralized P2P network, one that utilizes peer network capabilities to share project data with other network participants, therefore providing an alternative – and an improved – data distribution mechanism for volunteer environments.

In summary, the major contributions of this work include:

- *Analysis of current and new scientific volunteer computing applications' data requirements and restrictions*

- *Validation and application of distributed data management solution for BOINC and XtremWeb Desktop Grids*
- *Introduction of migration pathways for Service Grid data to Desktop Grid environments*

In addition to these major contributions, the following are notable contributions and achievements of this thesis:

- *Research provided the theoretical and prototype basis for concept development into production software*
- *Software and ideas contributed to further research by others at both Cardiff University and abroad*
- *Dissemination and development of concepts within the volunteer computing community furthered acceptance of P2P-based data distribution paradigms*

CHAPTER 2

Background, Motivation, and Related Work

Distributed computing, in one form or another, dates back almost to the beginning of the computer revolution. Local-area networks such as Ethernet were invented in the 1970s, and APRANET [2] introduced e-mail as far back as the early 1970s. In the 1980s, Usenet and FidoNet became popular and widespread ways to disseminate information and to support distributed discussion forums, with Bulletin Board Systems (BBSes) making them available to anyone with a basic computer and a modem. During the early 1990s, in my teenage years, I even ran a BBS, providing a forum for online multiplayer gaming, which allowed me and other locals to compete with long-distance individuals in many turn-based games (such as Risk) and tournaments. My BBS was a very good example of pre-(consumer-)Internet distributed computing, where game metadata were propagated throughout a network of BBSes and results were fed back to determine moves and battle calculations.

The Internet explosion happened in the years that followed and led the way for new kinds of distributed computing, ranging from websites processing client requests for information or consumer product orders, to remote control and monitoring of scientific applications and apparatus. Network speeds increased and it became possible to link computers together effectively in various ways for task distribution, including file sharing and scientific computation. This led to the emergence of a new distributed-computing infrastructure for science and engineering, termed “the Grid,” in the mid-1990s, followed by the adoption of public resource computing through the popular SETI@Home project in 1999, and, shortly thereafter, the widespread use of peer-to-peer (P2P) file-sharing programs such as Napster.

It is within the context of these three technological innovations that this thesis positions itself: the fusion of Grid computing, volunteer computing, and distributed (peer-to-peer) file sharing. This chapter gives an overview of these three technologies and their developments, as well as the driving motivations and use-cases for the development of my thesis to explore a distributed data architecture that can leverage P2P-based file sharing for distributed scientific computing.

2.1 Grid Computing

“The Grid” was not a new idea; rather, it was a coordinated push toward the realization of many distributed computing and integration concepts that have been around since the early Ethernet-invention era. Predictions of some of the promises of the Grid, such as easy connectivity and access to large amounts of distributed computational power, can be seen as early as 1965. It was then that designers of the Multics operating system envisioned the development of computational facilities into a system something akin to the electrical grid, where you simply plug in and get whatever resources you require.

These ambitious goals, of somehow being able to provide computational power as a raw resource, similar to electricity or water, have yet to be realized. Like power and water grids, computational grids will be built from the coupling together of many heterogeneous systems and organizations to form a massive system that is seen as a unified entity to the end user. Also, as in the development of integrated electrical systems, policies and standards will have to be created and deployed in order to guarantee interoperability between networks. Unfortunately for the development of computational grids, many of the similarities with electrical grids tend to end there.

Computers are much more complex than electricity or water, and the clients for the Grid are complex hardware and software applications that may have very specific, perhaps non-standard dependencies on other software or hardware. Figure 2.1 gives a high-level snapshot of what a Grid deployment might look like, with various different hardware, software, and scientific apparatus connected to it. When

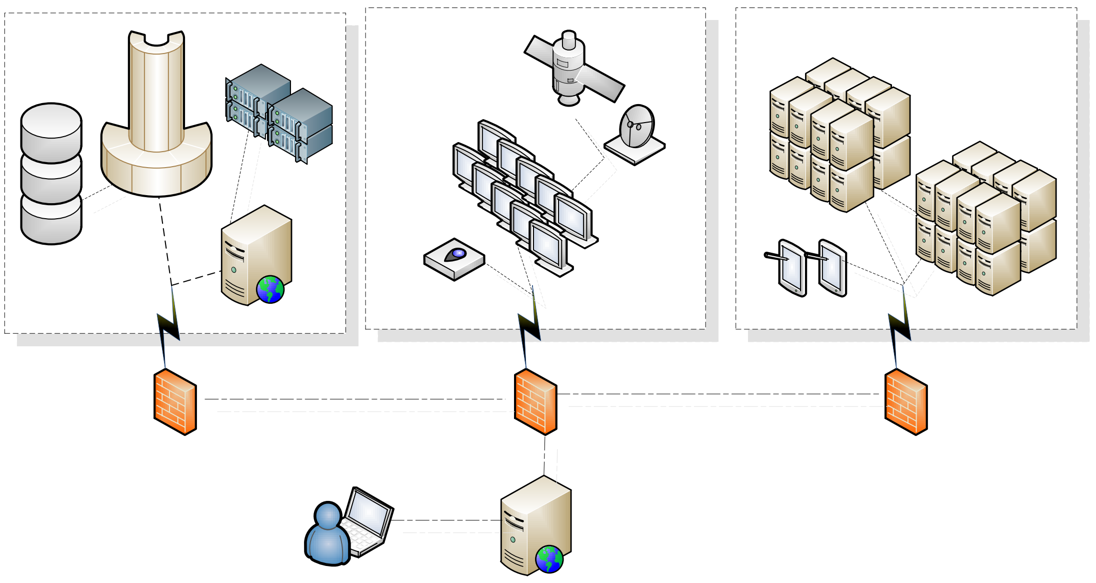


Figure 2.1: Typical Grid Computing Infrastructure — shows a typical infrastructure consisting of many heterogeneous resources controlled by multiple entities (i.e., organizations).

one looks beyond the basic technical issues of building a Grid, such as the deployment of networks and the integration of hardware and software installations, it becomes an even greater challenge to be able to build meaningful application middleware. To achieve the desired integration that would make the Grid able to provide CPU cycles and other resources as raw commodities, a software middleware is needed that will enable the developers of Grid-enabled applications to have a standardized way of communicating their needs to the resources, whether software or hardware, that rest on the Grid.

Although the term “Grid” has in the last decade come into widespread use and has become an accepted term in the larger community outside of academia and research, actually defining what it means proves to be a somewhat controversial and difficult task. Even though it is generally agreed that the Grid is the process of integrating distributed systems and institutions into one or many large virtual pools of resources, the meaning tends to change depending on who is using it and his or her agenda.

Ian Foster, largely accepted to be the father of the modern grid, and Carl Kesselman originally defined in 1998 what they saw as grids in terms of the infrastructure needed for connecting hardware and software together into large computational resources. Specifically, they state:

> *A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.* [3]

A mere two years later, Ian Foster and Steven Tuecke expanded this definition to make it more generalized and include the concepts of coordinated resource-sharing and problem-solving. They created the concept of virtual organizations, which put people and resources into dynamic groups with potentially complex relationships [4]. With this new definition, the Grid became something that was not confined to creating standards and building software to link resources together so they could be used in a more effective manner. Grid computing was expanded from a focused view on computational grids that provide high-end computing capabilities to include the forming of relationships and policies between sites and people and the coordination and enablement of resource-sharing and problem-solving.

In July 2002, Foster became more specific and gave a three-point checklist [5] for determining whether or not he deems something to be a grid. He defines a grid in terms of whether or not it is subject to centralized control, whether it uses standards and open general-purpose protocols and interfaces, and whether or not it delivers non-trivial qualities of service.

Foster, however, differentiates between something being a grid and the Grid. Whereas a grid could be anything that couples together distributed, non-centrally controlled resources and a common set of protocols and standards, the Grid must be a larger system in which all the individual resources are able to interoperate with each other through the use of specific standardized InterGrid protocols. In this scenario, one could be defined as a member of the Grid only if one followed the chosen standards and implemented the agreed-upon protocols. For example, if the Open Grid Services Architecture (OGSA) [6] had been selected as the official "Grid standard," one would be part of the grid only if one implemented the OGSA InterGrid protocols.

2.1.1 Conflicting Views

Although there is fairly widespread acceptance of Foster's second definition of the Grid, given in 2000, there has been some debate over his subsequent views. Members of industry primarily do business through the sale of proprietary closed software, and manage computational systems and resources varying greatly in size and scale. Thus they argue that defining a grid in terms of decentralization and whether or not it uses standard and open protocols loses sight of the true purpose of a grid. Further, by defining a grid in these terms, Foster is changing focus from what a grid actually does to how it is managed and implemented and what quality of service it provides; these should be attributes of a grid, not its definition.

In an article titled "Response to Ian Foster's What is the Grid," after sharing his views on how Foster's latest definition of a grid is too constrained and focused on the wrong things, Wolfgang Gentzsch, director of Grid Computing at Sun Microsystems, gives his own definition of what constitutes a grid:

> *A Grid is a hardware and software infrastructure that provides dependable, consistent, and pervasive access to resources to enable sharing of computational resources, utility computing, automatic computing, collaboration among virtual organizations, and distributed data processing, among others.* [7]

IBM was one of the major industry players in the development of Grid software and infrastructure in the early 21st century. By building middleware, portal frameworks, development tools, and other software infrastructure, it tried to try to take advantage of the widespread interest in Grid computing that prevailed at that time. Although it is working closely with Ian Foster and other members from the Globus Project at Argonne National Labs, IBM's definition of a grid, like Sun's, seems to follow from Foster's 2000 views and mentions nothing of his 2002 three-point checklist:

> *With grid computing you can unite pools of servers, storage systems and networks into one large system to deliver non-trivial qualities of service. To an end user or application, it looks like one big virtual computing system. The systems tied together by a grid might be in the same room, or distributed across the globe; running on multiple*

hardware platforms; running different operating systems; and owned by different organizations. [8]

2.1.2 Different Priorities

The differences in the definition and interpretation of the meaning of “the Grid” or “a grid” coming from industry and research can be attributed to the priorities and advantages that various groups find in Grid computing, and what they choose to emphasize.

On the one hand, academic institutes that are developing grid infrastructure and tools view the Grid mostly as something able to connect various large resources and special instruments. Protocols and standards are of paramount importance, as they determine whether these different resources will be able to interoperate. Since they are generally working in collaboration with other institutions and sometimes industry partners, the development of cross-institutional policies and agreements also plays a vital role.

On the other hand, the research labs and academic institutions that are deploying applications on grids are more focused on the capabilities of grid computing rather than the details. Although protocols and standards are important for the underlying infrastructure to be easily integrated into a seamless system, they are not the focal point of most application developers. They are more concerned with how the grid will actually improve the performance of their applications in an easy-to-implement manner and how it will grant them larger access rights and capabilities.

For those deploying applications on Grids, the primary concerns are performance and ease of use. What these developers need and want when building the grid infrastructure is a common language and set of APIs that will distance them from the underlying implementation details. This will allow them to focus on building their specific application codes, rather than on developing low-level grid infrastructure.

Private industry, being more directly profit- and product-driven, tends to view the Grid as a way to lower costs, increase margins, and enable new profitable product lines. By lowering the cost of large-scale computing and of developing an infrastructure that allows resources to be used at the optimal level, their overall operations costs will drop, producing more profit. In addition to lowering the costs of current operating procedures and technology, Grid computing promises to usher in a new era in which it is not necessary for companies and research labs to maintain their own resources (both hardware and software). Instead, it will be possible to farm out tasks to third parties, purchasing whatever resources are needed, when needed. This introduces a new market of consumers, which promises new revenue sources.

Industry's interest in Grid computing is not limited to being able to act as computing or storage farms. With Grid computing comes the demand for new types of software solutions at all levels, from end-user software portals to back-end load-balancing software that uses grid technologies to dynamically adjust resource utilization, not to mention all the hardware that various organizations are buying to build new grids.

The last major Grid user-group is the government. Different governments' views and definitions of the Grid will naturally vary depending on the country in question and its priorities. In the United States, one of the ways the Grid is being sold to the government is as a new way of combining the resources at national laboratories and research centers to build large systems that optimize efficiency and drive new areas of research that were previously impossible. The same is true for European collaborations, where deploying a grid infrastructure allows cross-national collaboration and aggregation of resources, as can be seen in the Enabling Grids for E-sciencE (EGEE), described in §2.4.1, and NordGrid activities. It is therefore my view that governments' view of the Grid closely resembles that of academics, with the exception that in the case of some government agencies implementing grid solutions, there may be very centralized control mechanisms in place and the underlying protocols may not be as open (e.g., defense industry grids).

The Grid as a more generalized economic catalyst also becomes a priority that is mainly exclusive to the government. Whereas private industry is more concerned with being able to push its individual products, governments look toward how they can use their vast resources to stimulate the economy in different ways. Thus the Grid has recently become an attractive candidate, with large amounts of funding put toward initiatives such as the German D-Grid, the EGEE, the Tera-Grid in the USA, and many other large-scale Grid infrastructure initiatives. For the past several years there has been a big push for the installation of high-speed networks that provide the needed interconnectivity. Also there is additional investing in high-tech innovations in the hope that they will promote larger economic growth and act as a stimulus to stagnating economies.

In the United States, it is also important to note the post 9/11 shift in focus toward homeland security. More emphasis is being put on the ability of the government to respond quickly to dangers arising from terrorist or other attacks and natural disasters. In this respect, the Grid is thought to be the answer to being able to quickly acquire the needed resources to perform whatever modeling is necessary, operating as an in-demand computational utility, much as Len Kleinrock suggested in 1969.

2.1.3 My View of the Grid

For me, the clearest single definition of the Grid, and what constitutes a grid, comes from Ian Foster and Steve Tuecke in 2000, where a grid enables coordinated resource-sharing and problem-solving in dynamic, multi-institutional, virtual organizations. For my own definition, I would expand upon this to incorporate part of Ian Foster's latest take on the Grid, in which he stresses that for something to be a part of the Grid, it must implement whatever inter-Grid protocols have been widely accepted as the standard.

However, I do not subscribe to Foster's checklist that defines what constitutes a grid. I view a grid as an abstract concept that should not be limited to implementation or policy details. It should be given the loosest definition possible that still provides meaning, inclusive rather than exclusive. The Grid, in contrast, should be something concrete, and in order to be part of the Grid, it is necessary that all members implement the same functionality that can facilitate interoperability. This requires grid members to agree to a well-defined definition of what being in the Grid means.

In the end, the Grid will need to be built on strict standards. Although extensions may be added by different groups to suit their individual needs (as we have seen in the Web world), there will be a base set of functionality that will exclude one from membership if it is not implemented in full. It is this base set of functionality, the standards and technologies that have been agreed upon to implement it, and the policies that enable it to happen that will eventually define the Grid.

2.1.4 Service Grids and Desktop Grids Compared

The Grid characteristics described in the preceding section refer mainly to what has since come to be known as a *Service Grid* (SG). Service Grids are generally comprised of large, coordinated sets of resources that work together to provide a large, integrated computing environment. The most successful Service Grids are collaborations between large national laboratories, universities, and other well-organized institutions with a high level of technical expertise. To join one of these grids, a non-trivial amount of resources is usually required; security arrangements must be made; a large set of complicated middleware needs to be installed, updated, and maintained by technically adept staff; and formal (legal) agreements are standard practice. For example, EGEE's production infrastructure, which provided about 40,000 CPUs, was composed of a relatively large set of 250 resource centers around the world.

To use the resources on a Service Grid, one must have valid security credentials as well as a resource allocation. Security is tight, with users normally being required to have an X.509 certificate issued from an accepted Certificate Authority (CA). This generally requires not only validating one's identity with a trusted partner, but also having a scientific affiliation as well as a legitimate problem to solve. EGEE was an extremely large Grid, with servers and research groups from all over the world joining it. It managed to provide access for about 200 different groups (virtual organizations, or VOs), consisting of a few thousand scientists all told.

Desktop Grids (DGs), in contrast, are generally composed of individual computers (i.e., "Desktops") that join together to provide an aggregate computing resource. Unlike Service Grids, which are based on complex architectures, Desktop Grids employ a simple architecture aiming to easily integrate a large set of (potentially) dispersed, heterogeneous computing resources. Desktop Grids are opportunistic, using "scavenged" cycles from otherwise idle computers. This paradigm represents a complementary trend concerning the original aims of Grid computing. In Desktop Grid systems, any number of potential contributors can bring resources into the Grid because the barriers to entry are very low: installation and maintenance of the software is relatively intuitive, no special expertise is required to join, and resources do not have to be dedicated or static.

Condor [9, 10] provided a specialized workload management system for CPU scavenging as far back as 1988. Created at the University of Wisconsin, the Condor developers saw the numerous “idle” computers around campus as potential resources for memory-light, compute-intensive jobs. The Condor project provided tools for grouping personal computers into pools of resources and paved the way for numerous Desktop Grid projects. These new Desktop Grid projects diversified from Condor’s goals of controlling, managing, and harnessing computational power from clusters and desktop resources by exploring (after the Internet explosion) a public-facing volunteer-based model, which aimed to take advantage of highly distributed desktop machines using a loosely coupled opt-in approach.

2.1.4.1 Resource Utilization and Workflow

Another differentiating factor between Service and Desktop Grids is that Service Grids have reciprocal agreements for resource utilization among partners, whereas participants in Desktop Grid systems cannot use the system for their own goals. In most Desktop Grids, the scavenged cycles are simply “donated,” with little or no expectation of a return. Complex brokering systems that proportion resource utilization and allocations based upon participation do not exist. Because of this, the Grid research community considers DGs as limiting solutions when it comes to resource sharing and solving issues such as load balancing and peak demand.

From a technical job-distribution and job-running standpoint, the main difference between SGs and DGs is the way computations are initiated. In Service Grids, a job submission or a service invocation is what initiates activity, using a *push model*, where the service requester pushes jobs, tasks, and service invocations onto passive resources. Once the request is received by the executing service or node, it becomes active and performs the needed activity. Conversely, Desktop Grids use a *pull model*, where resources with spare cycles request (i.e., pull) tasks from a repository, which typically is centrally located on a Desktop Grid server. In this way, resources play an active task-distribution role in a DG system — initiating their own activity based upon internal resource utilization and availability.

2.1.4.2 Exposure and Access

Both Service and Desktop Grids can be publicly exposed, or constrained to private domains. A public grid refers to a grid that connects resources from different administrative domains, is typically interconnected by a wide-area network, and has a facility for new non-affiliates to join. Private (typically local) grids, in contrast, mostly connect resources from within the same administrative domain using a local-area network or a Virtual Private Network (VPN). The European Grid Initiative (EGI) [11], the Open Science Grid (OSG) [12], and the TeraGrid [13] are examples of large, public Service Grids, whereas the Oxford Campus Grid [14]) exemplifies a non-public Service Grid interconnecting local clusters. A good example of a local Desktop Grid is a university lab with a Condor pool.

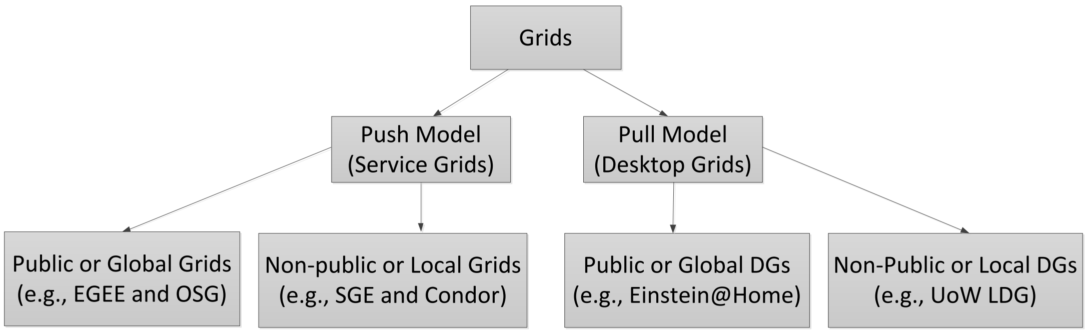

Grids
Push Model (Service Grids)
Pull Model (Desktop Grids)
Public or Global Grids (e.g., EGEE and OSG)
Non-public or Local Grids (e.g., SGE and Condor)
Public or Global DGs (e.g., Einstein@Home)
Non-Public or Local DGs (e.g., UoW LDG)

Figure 2.2: Taxonomy of Grid systems — shows a Grid taxonomy from the Desktop Grid point of view.

Within Desktop Grids, two general categories emerge, as shown in Figure 2.2. First, there are large-scale public Desktop Grids, with open participation policies, and second, there are non-public or local Desktop Grids that are often comprised of institutional resources. Local Desktop Grids can be broken down further into volunteer and, for lack of a better word, “involuntary” Desktop Grids. In an involuntary DG, individual desktop owners are instructed to contribute their resources. Examples are the University of Westminster’s (UoW) DG and the Extremadura School DG. The Extremadura School DG is a public Desktop Grid in which the regional government (in Spain) instructed the schools of the region to contribute their desktops to the system. Likewise, the University of Westminster DG consists of university lab computers that have been joined together to provide a several-thousand-node computing resource for the university.

Volunteer Desktop Grids usually solicit the general public to donate resources, so called “Public-Resource Computing,” and would be classified as *public* Desktop Grids. They have been extremely successful. For further discussion of the volunteer computing movement and the software and projects involved, see §2.3. It should be noted that all the previously mentioned Desktop Grid systems are centralized, with a common point of reference for job assignment and result aggregation; however, there are some decentralized Desktop Grid systems, with the OurGrid DG infrastructure from Brazil being a perfect example [15].

2.2 Cloud Computing

The scalability, underutilization, and load-balancing issues that helped promote Grid computing also led to another new form of highly successful and popular distributed processing called “Cloud” computing. In the years after the dot-com bubble, Amazon played a key role in the development of Cloud computing when it modernized its data centers, and, similar to many computer networks, had to build to meet peak demand (such as the holiday season, or other large volume times), leaving much of its capacity idle for long periods of time. Seeking to leverage this opportunity, Amazon released a new product in 2006, Amazon Web Services (AWS) [16], that allowed external customers to use these idle cycles. More developments followed (Eucalyptus [17], OpenNebula [18]), leading to the ability to run a full-fledged virtual machine (VM) on a third party’s resources, and the “Cloud Computing” movement had begun.

“Cloud computing” is a term used to describe the latest incarnation of on-demand, scalable network computing. In Cloud computing one sees the computational and storage resources, and more recently even services themselves, as black boxes to one’s application. Complex and simple services can be hosted in the Cloud, and since a Cloud is really just a highly scalable infrastructure for hosting practically any software program with any network-contactable service endpoints, what constitutes an individual Cloud is at the complete discretion of the service provisioners. Figure 2.3 shows the versatility of the Cloud, where everything is thrown in, ranging from application servers and distributed databases to program code.

This new computing paradigm has become possible due to advances in virtual machine technology. Virtual machines allow all the operating systems themselves to be treated as applications running on an operating system. Therefore, just as one might have multiple browser sessions open, one can have multiple operating systems open. It is even possible to have virtual operating systems running within a virtual operating system, albeit with an additional performance cost.

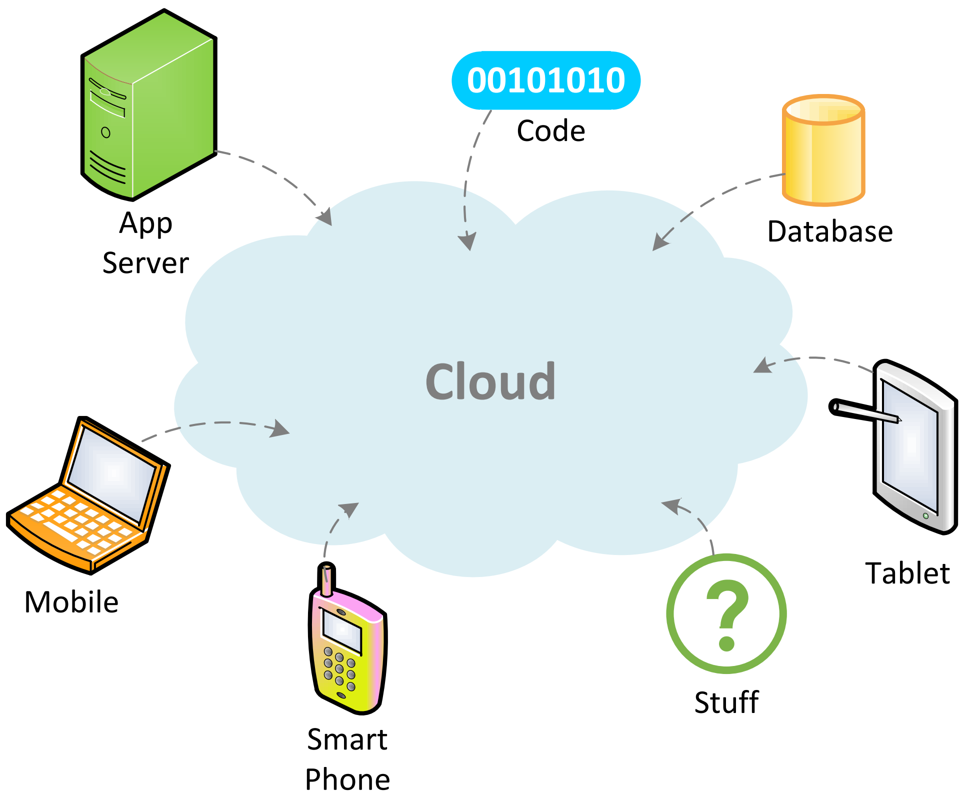


Figure 2.3: Abstract View of Cloud Computing — gives a high-level view of Cloud computing, where everything is thrown into “the Cloud” and is accessible as independent resources. Clouds can host individual programs, such as in Microsoft’s Azure, or full-fledged operating systems, as in Rackspace and Amazon’s S3. No common service interfaces or standards exist for the Cloud or the services that are deployed into it.

2.2.1 Flexibility and Scalability Benefits

Having the operating system itself as an application opens up a new range of possibilities. If the entire operating system, and everything installed within it, can be represented as a single file on another operating system, one is now able to backup, restore, easy clone, and start multiple instances over a completely configured system and dependent applications, all in an automated and scalable fashion. This is where Cloud computing comes in; Cloud computing recognizes that because multiple operating systems can now be run as virtual machines on one piece of physical hardware (i.e., one machine), those configurations can themselves be fine-tuned to whatever the task requirements might be, and then turned on and off on demand. For example, with virtualization and Cloud computing, it is no longer necessary to purchase an entire physical machine at a Web-hosting company to run one's website. Rather, one can simply purchase a virtual machine that is running "somewhere." Perhaps a dozen other virtual machines are running on the same piece of hardware; it doesn't matter as long as the quality of service that one is paying for is being provided.

Since each virtual image instance might use only a fraction of the real computational power and storage available, in the event that more storage or more computational power is needed by a particular instance, a new hard disk does not need to be purchased and new memory does not necessarily need to be inserted because a physical limitation has been reached. This is because a particular virtual machine needing more resources can simply be reconfigured to use a larger proportion of what is available, and if and when the physical hardware reaches its limit, a subset of the VMs running upon it can simply be transferred to new physical hardware.

A new kind of dynamic resource distribution was born. No longer does hardware need to be tightly coupled to the software running on it. For computation, it is therefore no longer necessary to know where something is running; instead, one can simply take comfort in the fact that it is running. The same applies to data. In a Cloud computing paradigm it is not necessary to know the location of data, only to be certain of their existence, availability, and the quality of service that can be expected when trying to access files.

2.2.2 Cloud Computing as a Business Model

Many large businesses have become heavily involved and invested in Cloud computing. This includes not only the big names in the consumer software market and online industry, such as Amazon, Google, and Microsoft, but also many companies dedicated only to Cloud computing, such as Rackspace. The large corporate shift toward Cloud computing is much more concentrated than the commercial backing of previous distributed computing systems. The primary reason for this shift is that Cloud-based technologies can be widely sold as useful service hosting to a wider audience than traditional Service or Desktop Grids. Companies investing in the Cloud see the potential of not only leasing storage and CPU cycles, but also providing value-added services that can be easily customized or replicated to new customer bases. For example, Amazon's Simple Storage Service (S3) [19] provides a seemingly endless amount of storage to anyone willing to pay for it, and this can prove beneficial for scientific applications, as explored by Palankar and colleagues [20], albeit with a direct fiscal cost. And when Amazon needs more storage space, it simply adds new servers to its data farms. At no point does the end-user need to worry about expanding in-house data capacity or buying new disks.

The same scalability principles apply to service hosting. For example, Microsoft's Azure service allows programmers to write .NET applications and deploy them to their cloud service layer. Azure provides more than just a flat operating system package. It also provides an entire .NET hosting environment in which users are able, for example, to store data in a distributed SQL database. And unlike a traditional database for which, as demand increases, a larger and more powerful machine would be needed to handle the requests or a distributed system would have to be employed, Azure's SQL service does this automatically, scaling dynamically based upon demand. The ability to easily scale can be very beneficial to companies using these services. By offloading scalability to someone else in the Cloud, businesses no longer need to worry about overbuying hardware and software to meet peak demand.

2.2.3 High Performance versus Highly Scalable Computing

Many might feel that the recent success and rush toward Cloud computing make it the next evolutionary step after Grid computing, supplanting it for large computing needs. This would be a false assumption because one must differentiate between high-performance computing (HPC) and highly scalable computing (HSC). Cloud computing provides a highly scalable environment, where resources can be brokered on demand and consumption can easily scale to meet problem size. This is an excellent system for disparate, embarrassingly parallel problems and computation, such as many consumer-facing applications — for example, websites, credit card processing, and parallel data processing.

What neither Desktop Grids nor Cloud Computing can provide is a high-performance computing environment. Not every problem can be broken into discrete computing segments that can be independently computed over long periods of time without requiring network processing and synchronization. Programs that rely on the Message Passing Interface (MPI) [21, 22] are typical examples of applications that would perform very poorly on Cloud computing resources [23]. MPI was designed as a way to tightly integrate separate memory spaces by providing a high-throughput message passing interface for calculations to communicate and coordinate. MPI seeks to emulate what a shared-memory system would be able to provide, but with a greater latency and the requirement that “messages” are sent between nodes rather than via a shared memory space that can be accessed by all. The advantages of using MPI are that an application can be deployed on a relatively cheap cluster and run on many hundreds of processors, which “act” as if they are a single application and computer. A shared-memory system of similar scale would be tremendously expensive, if not impossible, to create. Most ($\approx$83%) of the Top 500 [24] supercomputers these days are actually clusters, with the vast majority of the rest being closely related, massively parallel processing (MPP) computers, leaving symmetric processing (SMP) machines, Constellations, and Vector computers mostly in the past.

Despite being built primarily with some of the same fundamental building blocks as Clouds (i.e., mass market CPUs), Grids provide a very valuable resource for High-Performance Applications that Clouds currently do not. Due to their high-speed interconnects, Grids have fast disk access and interprocessor communication facilities that are lacking in even the best Cloud environments. I see it as very unlikely that the Cloud will suppliant Grid capabilities in the near future for HPC, given the fractured nature of Cloud operations and their general lack of dedicated high-speed interconnects such as InfiniBand, Gigabit ethernet, and, in the not-so-distant past, Myrinet.

Service Grids therefore have a different target group than Cloud computing, one that needs high-performance computing, not just highly scalable computing. Desktop Grids, however, are very similar to Clouds, in that they are providing a high-latency environment. Indeed, most Desktop Grid applications could be run on Cloud resources, and there is a push to convert much of the DG infrastructure to rely on virtual machines. Commercial Clouds, however, still prove to be an order of magnitude more expensive than the highly successful “volunteer computing” Desktop Grids that preceded them [25].

2.3 Volunteer Computing

Volunteer Computing, also referred to as “Public-Resource Computing,” is a relatively recent phenomenon that surfaced with the broadening of the consumer

Internet. In the mid-1990s the Search for Extraterrestrial Intelligence (SETI) [26] project was collecting very large amounts of radio data from radio telescopes in Mexico. Each of the telescopes pointed into outer space and listened for narrow-bandwidth radio signals (not known to occur naturally) that might have been transmitted (albeit thousands if not millions of years ago) across the cosmos by some form of intelligent extraterrestrial life. The problems that arise when searching for these signals – beyond the basic question of whether or not they exist – are, first of all, the vast distances of space and, later, the Earth's atmosphere. As radio waves are transmitting through space, they can come in contact with interference (e.g., radiation, and solid objects such as stars and planets); after they finally reach Earth, a large amount of interference can be picked up by the telescopes due to the Earth's atmosphere and other environmental considerations. It can therefore be very difficult to distinguish between what might be an interesting wave from an advanced civilization saying "Hi" and artifacts and other "garbage" that might be corrupting the signal. Computer models using advanced algorithmic analysis can be applied to the data to "clean them up" and to differentiate signal noise from interesting signals. However, these models can require vast amounts of computing resources, not necessarily because a given analysis takes a large amount of processing time, but rather due to the vastness of the data itself.

With large numbers of computers becoming connected to the Internet in the 1990s, and in line with the thoughts of Distributed.net [27] (est. 1997) and the Great Internet Mersenne Prime Search (GIMPS) [28] (est. 1995), SETI researchers had an idea: What if some of the millions of people who now owned personal computers could be convinced to install a non-intrusive program that would look like a screensaver but would in the background process radio-telescope signals? Since SETI data did not need to be analyzed in parallel on an expensive Service Grid machine, the data sets could be split into discrete data partitions, each of which could be independently analyzed to see whether any interesting signals arose. Furthermore, due to the loosely coupled nature of the computation and the data being relatively unrelated, it was not even necessary that certain data elements be analyzed before or after others. SETI split their data analysis into a so-called "bag of tasks," which would then be distributed among many thousands, and even tens of thousands, of independent computing nodes, as shown in Figure 2.4. This was done through a newly created software project to coordinate task distribution and result aggregation entitled SETI@Home [29, 30].

SETI@Home was extremely well received. It provided contributors with feed-

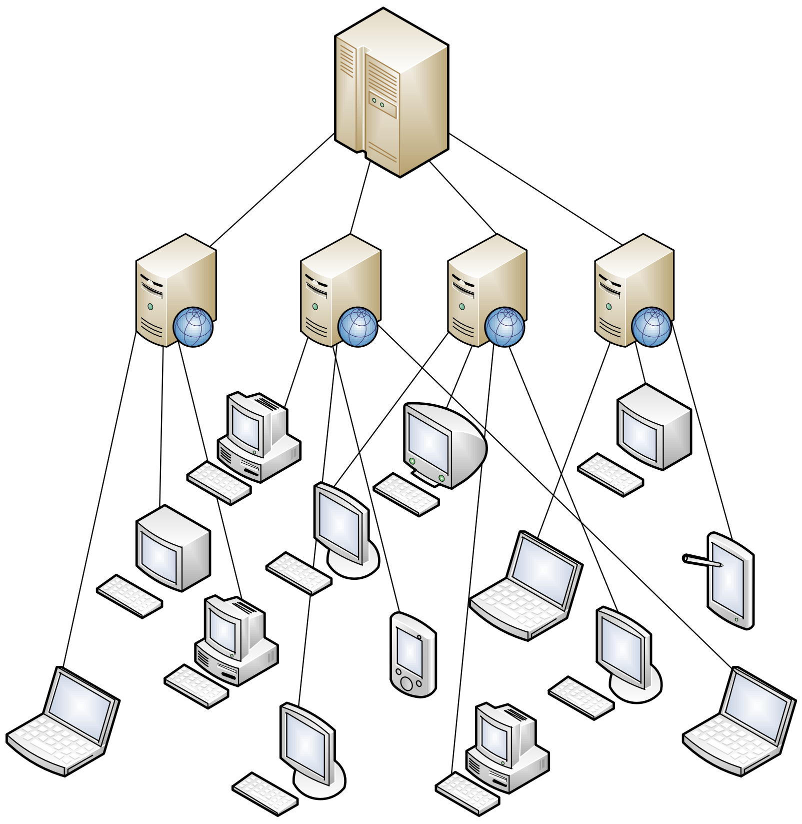


Figure 2.4: Volunteer Computing Workflow — gives a diagram showing how a a central project sends work to a much larger number of distributed resources for computation.

back, through a screensaver, that engaged the public and let them directly participate in a huge scientific experiment. Tens of thousands of people donated their computing power and became part of what is termed “public distributed computing” or “public resource computing” [31], in which jobs are executed by privately owned and often donated computers. SETI@Home had created, within a very short time-frame, for the purposes of raw computational power, the largest (distributed) supercomputer in the world, with processing power far exceeding anything that could be built and maintained in the lab. The project was so successful that the SETI@Home software was rewritten into a general framework to allow other scientists to do the same thing: the Berkeley Open Infrastructure for Network Computing.

2.3.1 Berkeley Open Infrastructure for Network Computing

The Berkeley Open Infrastructure for Network Computing (BOINC) [32, 33] expanded beyond organizational boundaries and became the largest and most successful volunteer computing application to date, taking advantage of unaffiliated and donated computing resources. In the years that followed its release, more than 50[1] distinct scientific projects adapted their applications to run on BOINC, and almost three million total computers from over 200 countries have registered to date [34]. The scientific applications making use of BOINC are very diverse, ranging from NASA's Climate@Home [35], which focuses on long-term climate prediction, to Einstein@Home's [36], aimed at detecting certain types of gravitational waves. Although these projects may be diverse in their scientific nature, they all share a common thread: they have scientific problems that can be split into discreet work units that can be computed in parallel in a highly distributed and volatile environment, and they are CPU-intensive, with a large ratio of CPU cycles relative to data transfer requirements.

The BOINC software stack contains a scheduling server and a client program that are installed on users' machines. The client software periodically contacts the scheduling server to report its hardware and availability, and in response receives a set of instructions (i.e., "work unit") for downloading and executing a job (see Figure 2.5). After a client completes the work unit, it uploads resulting output files to the scheduling server and requests more work. This disconnected nature of work generation, distribution, and result aggregation makes BOINC especially well suited to the volatile nature of volunteer computing networkings. To adapt an application to BOINC, a new project must not only prepare its data and executable code to work with the appropriate libraries and client/server infrastructure, but also set up and maintain its own individual servers and databases to manage the project's data distribution and result aggregation.

The BOINC server is the key part of a BOINC-based Desktop Grid. It provides an entry point for the users; stores applications, their related work units and user information; and deals with requests from BOINC clients. BOINC servers use a Web server (e.g., Apache) for the project users, which exposes a simple Web page offering basic functionalities: user registration, statistics, query, and BOINC client download requests. The BOINC server also operates as a user forum related to the project, where users can ask questions and report their problems.

[1]There are over 50 known BOINC projects. At the time of this writing, the BOINC website has a list of 25 with which they have been in direct contact: http://boinc.berkeley.edu/projects.php

BOINC uses a relational database (MySQL) for storage of applications, their related work units, as well as client and user information.

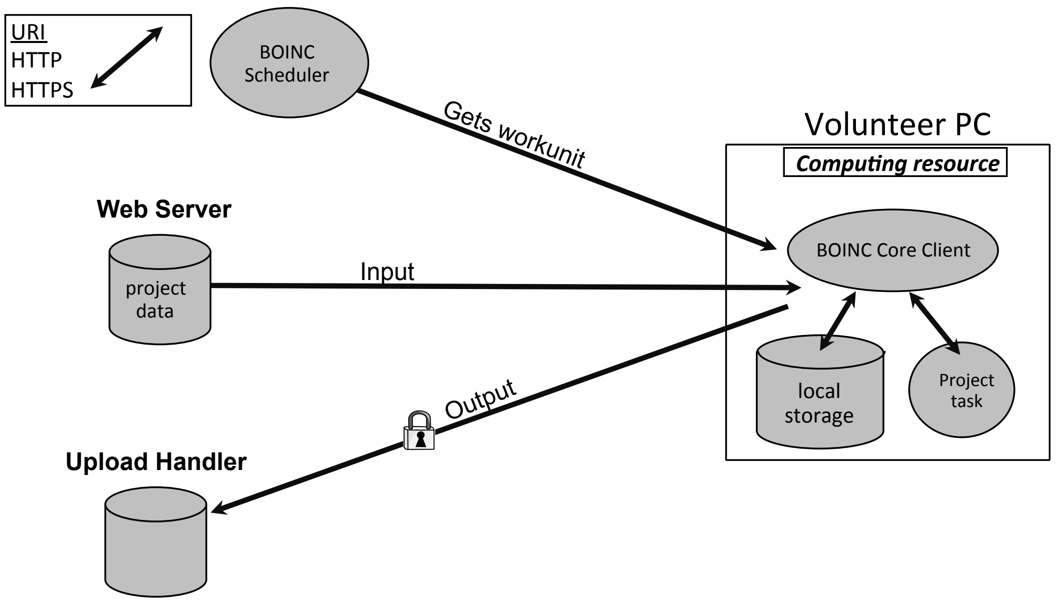


Figure 2.5: BOINC Architecture — showing how the different components interact over a wide-area network.

For data distribution, BOINC projects generally use a single centralized Web server or, in the case of more popular projects, a set of mirrors. In the current release of the BOINC client software, standard HTTP requests are used to download input files. This centralized architecture, although very effective, incurs additional costs and can be a potential bottleneck when tasks share input files or the central server has limited bandwidth. This can lead to additional costs on BOINC projects and can quickly become taxing on hardware resources, as well as inefficient, especially as replication factors increase and many tasks begin to share the same input files. Projects can add more mirrors to accommodate increased loads; however, this puts extra administrative burden on the project organizer, incurs a network cost, and can prove very time-consuming to manage.

BOINC uses a centralized architecture for task management and data distribution. Each project that uses the BOINC middleware is responsible not only for setting and maintaining its own entire infrastructure, but also for attracting volunteers to perform the computations. Although this has been an extremely successful approach for well-known and popular projects, it fails to provide an effective model for individual researchers to leverage volunteer computing. Seeing this gap, other projects have been developed that create a “community computation" paradigm, where the users of the infrastructure are also the volunteers. In this scenario, scientists donate their idle cycles to the community and are later afforded the ability to leverage the community resources when they need to perform a particular compute-intensive task. OurGrid and XtremWeb are examples of this more *ad hoc* "peer-to-peer"-based volunteer computing.

2.3.2 XtremWeb

XtremWeb [37], like BOINC, is a software system that gathers unused resources on donated computers and makes them available to scientific projects. Unlike BOINC, which has centralized servers and in which users subscribe to a relatively static set of projects for which they will donate their CPU time, XtremWeb allows multiple users and multiple (changing) applications to run concurrently on the system.

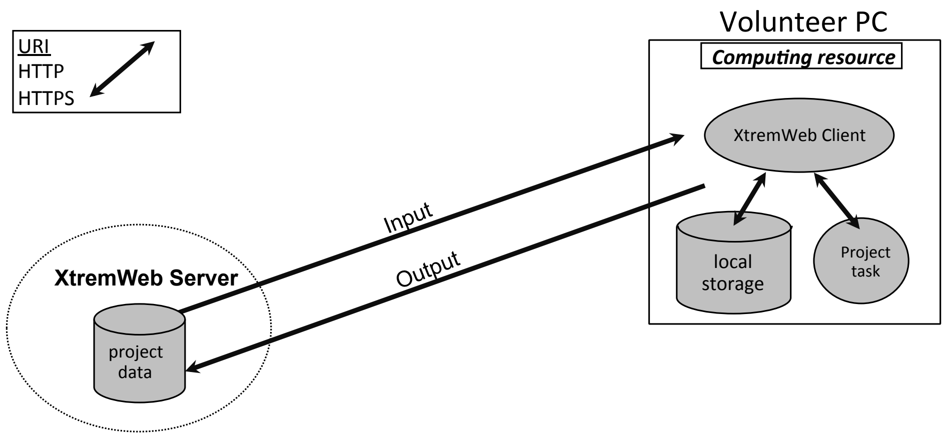


Figure 2.6: Data Access in XtremWeb — shows data access in XtremWeb, a Desktop Grid computing platform.

XtremWeb provides a platform in which scientists and volunteers can share their respective resources with one another; it is often the case that providers of resources are also the users of the system. If one were to think of BOINC as a large, stable, managed, and centralized volunteer computing system for distributing workloads, XtremWeb would be its smaller peer-to-peer cousin that organizes resources in an *ad hoc* manner and allows the running of arbitrary code, provided it has been signed and verified by a trusted party. Similar to BOINC, work units in XtremWeb are provided with the URIs of input files, as shown in Figure 2.6. These are downloaded as a preprocessing step when a client job is launched.

2.3.3 Volunteer Computing Application Profiles

Chapter 1 briefly introduced how the volunteer computing platform has become a popular means of providing vast amounts of processing power to scientific applications through the use of personal home computers. Many projects have successfully taken advantage of the BOINC middleware to garnish vast amounts of computing resources for their data analysis needs. To date, with little exception, the projects using BOINC have had a very large CPU processing time to data transfer ratio, thereby allowing a significant amount of offline computation with a relatively low data-transfer cost.

In this section, three existing highly successful volunteer computing applications are profiled, to set the stage for discussing the needs, limitations, and future of data delivery for volunteer computing networks. Each of these projects represents a variance in current BOINC-based applications. The first is SETI@Home, the original BOINC project, which has relatively small work-unit sizes, at 340 KB each. The second is Einstein@Home, a highly successful project analyzing gravitational wave data. Einstein@Home requires $\approx$6.5 MB of data for one work unit. The last application examined is Climateprediction.net, which has even larger data input requirements with $\approx$200 MB per work unit.

2.3.3.1 SETI@Home: Searching for Alien Life

The SETI@Home project, introduced in Chapter 1, is analyzing data from large radio telescopes, hoping that signals or patterns containing messages from extraterrestrial life can be identified. SETI@Home is the project that started the BOINC middleware and therefore has the longest history of using BOINC. It is arguably the most successful and well-known volunteer computing project to date. It fits very well into a volunteer computing distributed data paradigm, having relatively small work-unit sizes and a relatively long average processing time of two hours on a CPU (15–20 minutes on a fast GPU); see Table 2.1.

The data needs of the $\approx$233,000 active users require project servers to provide

Table 2.1: SETI@Home Project[a] Overview

SETI@Home				
Replication	**Task Size**	**Upload Rate**	**Processing Time**	**Tasks Per Day**
2	340 KB	50 MB/s	≈2 hours	≈1,000,000

[a] Does not incorporate initial 3 MB download of the software that occurs when each client joins the project, or when the software is updated approximately every six months.

≈50 Mbp/s of consistent upload, which equates to serving over 16 Petabytes a month. To provide result checking and ensure job validity, each SETI@Home task is replicated to two unique workers. Input data are not shared beyond the two workers that analyze a task, meaning that data requests to project servers could theoretically be cut in half if files were shared among network participants. The current throughput of SETI@Home as a computational platform is an impressive 550 Teraflops, putting it on par with many of the top supercomputers in the world.

2.3.3.2 Einstein@Home: Exploring the Fabric of the Universe

In early 2005, Einstein@Home [36] began using BOINC to distribute data collected from the LIGO detectors [38] and Arecibo Observatory's radio telescope (see Figure 1.1) to tens of thousands of volunteer computers to search for continuous gravitational waves. Six months after its launch, Einstein@Home was already producing 20 teraflops of sustained computational power, and was transferring half a terabyte in data each month to serve demand. At the time, that represented the equivalent of a 20-million-dollar supercomputer with a seven-thousand-dollar-per-day electrical bill. However, these were all donated resources, and the direct cost to the SETI@Home project was only a small fraction of this amount. Seven years later, at the time of this writing, Einstein@Home's processing power has increased to an impressive 250 teraflops. This puts its processing power on par with many of the top 50 supercomputers in the world.

Einstein@Home currently performs three different data analysis tasks. The first is the gravitational wave search that started Einstein@Home. This search analyzes collected data for constant, monochromatic, gravitational waves, such as those that might be given out by celestial artifacts such as rapidly spinning non-axisymmetric neutron stars. These signals do not change very much in their amplitude or frequency. Therefore, they can be searched for and detected more easily than gravitational waves from supernovae or neutron star and black hole mergers, which would have higher amplitudes and require more complicated analysis. Typical jobs require four to five hours on a CPU to compute about seven megabytes of data. To conserve bandwidth and processing power, each job is initially sent to only two different hosts to verify results (i.e., form a quorum), with a third host being used when one of the original results is not returned on time or when the quorum disagrees. The overall failure rate of a computational task, due to download errors, host errors, or other problems, is ≈6%, of which only ≈0.1% is due to quorum disagreements. Currently, the continuous gravitational wave search processes ≈75,000 results per day, representing a total data-load of over 15 terabytes a month in combined outgoing data transfer.

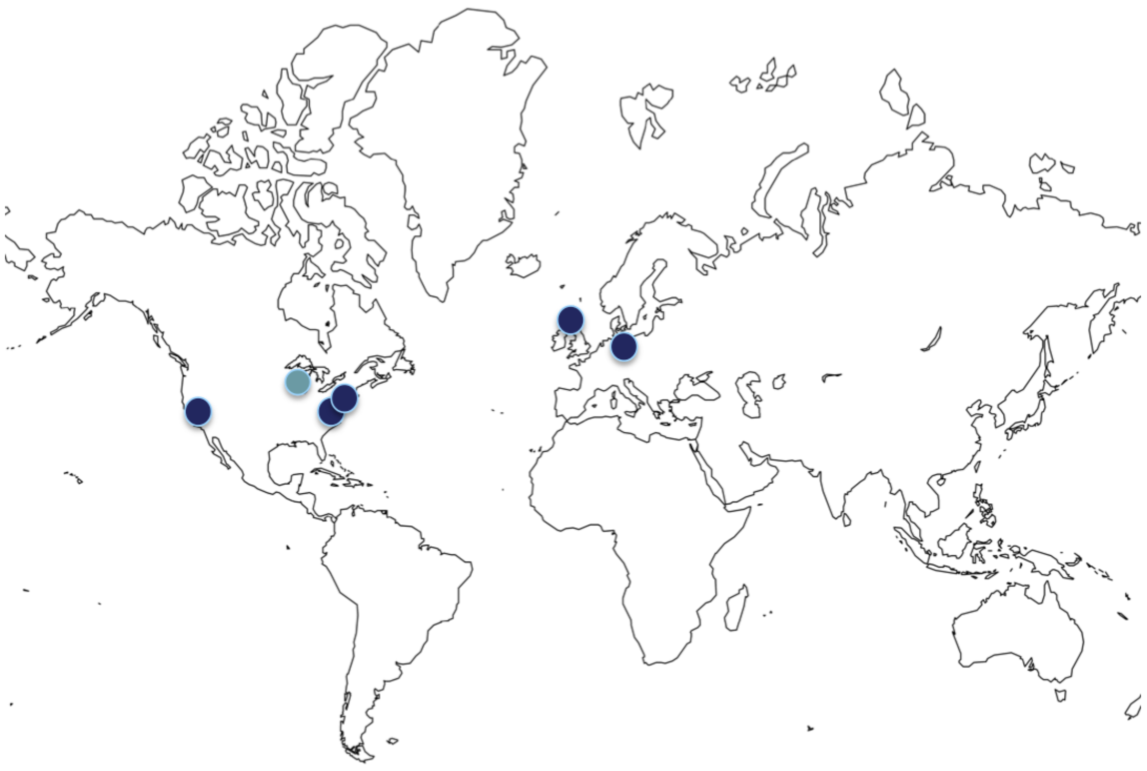


Figure 2.7: Mirror Locations for Einstein@Home — shows how Einstein@Home achieves data distribution through full mirroring of input data to five geographically distributed sites around the globe, in addition to the original data source. Data originate at UWM, and the five partner sites are part of the LIGO Scientific Collaboration (LSC).

Similar to other BOINC projects, Einstein@Home mirrors its input files to several geographically distributed sites to provide data scalability, as shown in Figure 2.7. One problem with this centralized approach is that as additional data sources come online, higher-resolution data become available, or compute/data ratios change due to increasing processor speeds, the data-transfer needs can increase substantially. The current solution to this problem is to increase network speeds continually, add more mirroring servers, and/or down-sample and preprocess the data. In addition, the Einstein@Home team has invented a procedure they call "locality scheduling," where tasks are preferentially assigned to machines that have already downloaded shared input files in a previous simulation. However effective they might be in the short term, these methods necessitate additional logistical work and do not provide a dynamic solution that doesn't require direct human intervention and resource allocation.

The other, newer analysis in the Einstein@Home project is pulsar searches: one in radio and one in gamma-rays. For these searches, the raw data are stored at the Albert Einstein Institute (AEI) in Hannover, rather than University of Wisconsin - Milwaukee, where Einstein@Home gravitational raw data are located. Unlike the gravitational wave analysis, the data for the pulsar searches are not mirrored. The gamma-ray analysis does not need mirroring or any complicated data management solution because the input files are small, less than one megabyte each. For the binary radio pulsar (BRP) search, the data are not mirrored because they are not reused very much; and the act of mirroring itself would require the data to be transferred at least once, thereby significantly reducing any benefit except to reduce peak demand.

Table 2.2: Einstein@Home Project[a] Overview

Einstein@Home				
Replication	**Task Size**	**Upload Rate**	**Processing Time**	**Tasks Per Day**
Gravitational Wave Analysis				
2	6–7 MB [a]	(5 mirrors)	≈5 hours	≈75,000
Binary Radio Pulsar				
2	≈32 MB[b]	30 MB/s	≈40 min. (GPU)	≈700,000

[a] Does not incorporate initial 40 MB download for each new client joining project.
[b] Once or twice a year, for a period of approximately two weeks; task size increases as the end of a run is reached and analysis needs to be done on data that are spread over the whole frequency range, which initiates a ”full sized” download for each task.

A BRP search can effectively utilize a computer's GPU, completing an analysis of 32 megabytes of data in $\approx$40 minutes. This faster processing time of larger data sets has created a significant increase in data demands. Whereas the gravitational wave search consumed approximately one megabyte of data every 30 minutes, the BRP search can consume almost a megabyte per minute. This is set to increase as the analysis code is optimized even more for GPU processing.

Currently, the two pulsar searches are fed (outgoing) from a single Web server, equipped with a 16 disk RAID-6. To serve the pulsar data, around 700,000 downloads a day, an average bandwidth of $\approx$30 MB/s is required, with a peak of over double that at 70 MB/s. *Apache* had been used in the beginning to serve HTTP data, but quickly hit a bottleneck. The Einstein@Home team then switched to *lighttpd*, which alleviated the problem for a short time, before finally grinding to a halt as data requests increased beyond its capacity. The latest solution is to use *nginx*, the Web server that powers Facebook, Git, and Sourceforge. Currently Nginx is able to satisfy the team's requests; however, it is exploring other distribution options to satisfy future and growing demand.

2.3.3.3 Climateprediction.net: Predicting the Future of the World's Climate

Climateprediction.net (CPDN) [39] is a parameter sweep application that investigates uncertainties of climate model input parameters. By reducing the likelihood that the input variables are flawed, CPDN is able to help climate models achieve more accurate results. CPDN does this by running hundreds of thousands of different models (i.e., a large climate ensemble) and looking at divergence patterns and how the model output varies when given slight changes to various physics parameters. By analyzing these changes, CPDN is able to provide a better understanding of how the models are affected, which can be used to fine-tune the models and help them provide more accurate predictions.

By leveraging the power of volunteer computing, CPDN was able to achieve, within a short time after launching in September 2003, a larger processing capacity than the Earth Simulator in Japan, which at the time was the world's largest climate modeling facility. Currently CPDN has an average performance of $\approx$35 Teraflops, with over 250,000 total users, 20,000 of whom are active on a regular basis.

The project is run primarily by Oxford University in the United Kingdom. Climate Prediction.net is currently composed of two experiments: Weather At Home

Table 2.3: Climate Prediction.net Project Overview

Climate Prediction.net				
Replication	**Task Size**	**Upload Rate**	**Processing Time**	**Valid Results** [a]
Weather at Home				
3	131 MB	≈3 mb/s	≈4.5 days	≈32%
HadCM3N				
2	55 MB	≈3 mb/s	≈31 days	≈14%

[a] Unlike most other BOINC projects, CPDN puts no time limits on work units, and given the nature of the parameter sweep, it is not imperative that every work unit is computed, as it would be in a data analysis project such as Einstein@Home. The lack of time limits and extremely long processing times of work units contribute to the extremely low rate of valid result return.

and HadCM3N. Data are managed by seven upload servers, each providing on average 0.39 MB/s, or an equivalent total of 234 GB per day. The project's data requirements and throughput are given in Table 2.3. One aspect that distinguishes CPDN from other BOINC projects is that the output file size is significantly larger: for the Weather At Home experiment, output data are 74 MB, and for HadCM3N they are a significant 215 MB.

2.4 Service and Desktop Grid Interoperability

Service and Desktop Grid environments are separate entities, with different infrastructures, APIs, user bases, and target application types. When the research supporting this dissertation began, there existed no way to leverage the individual advantages of one environment over the other. That is, there was no unified Service and Desktop Grid environment, and there were no bi-lateral migration paths between the two. Combining these two distinct computing platforms would enable researchers to use the well-known Service Grid infrastructure, while at the same time having the availability of Desktop Grid resources when they wanted to run an "embarrassingly parallel" application that did not require the tightly coupled Service Grid systems. Achieving a unifying solution for these two different environments required overcoming a number of difficult technical hurdles, such as minimizing or eliminating the high overhead of setting up a Desktop Grid project and finding a way either to harmonize or work around the different security infrastructures.

Interoperation between Service and Desktop Grids had been explored before, for example, in the Lattice project [40] at the University of Maryland, the SZTAKI Desktop Grid [41] in Hungary, the Condor project at the University of Wisconsin, and the Superlink project [42] at Technion. These various projects looked at the different aspects of combining Service and Desktop Grids, yet didn't seek to provide a user-driven infrastructure that allowed on-demand submission of jobs from Service to Desktop Grids. For instance, the Condor project implemented a mechanism for backfilling Desktop Grid jobs into Service Grids when peak demand was not met, thereby ensuring the Service Grids were fully utilized.

One far-reaching goal of fusing SG and DGs is the notion of combining job execution between the two environments into a single computing infrastructure that can be leveraged by everyday scientists. When looking for a complete Service-Desktop Grid interoperability solution, the European Union Enabling Desktop Grids for e-Science (EDGeS) project's goal of providing job migration between the Enabling Grids for E-sciencE (EGEE) Service Grid and BOINC and XtremWeb volunteer computing platforms is noteworthy. It comes the closest to full interoperability among the various projects that have attempted to combine the two systems.

2.4.1 Enabling Grids for E-sciencE (EGEE)

Enabling Grids for E-sciencE (EGEE) [43] was a very large and well-funded[2] European Union project. It followed in the footsteps of the EU DataGrid to build on advances in Grid computing technology and provide a robust Grid infrastructure for research. It was originally entitled "Enabling Grids for E-Science in Europe," but was later changed (dropping the "Europe") to reflect its more global orientation. During its reign, the EGEE was the largest Grid infrastructure in the world, lasting from 2004–2010, under three successful and consecutive rounds of funding proposals. Discussing the nature of EGEE is very relevant to the research presented here because it was the main Grid environment active during the formulation of the solutions and ideas presented in Chapters 3 and 4. It set the stage

[2]The EGEE was funded in three phases. The first lasted from 2004–2006, the second from 2006–2008, and the third from 2008–2010. The total funding for the three EGEE projects exceeded 130 million Euros.

of much for the data requirements that Service to Desktop Grid migration might present.

EGEE connected many different institutional clusters and supercomputers from around Europe (and later globally) into a single Grid entity with common elements, such as storage, job queues, and a shared security infrastructure. For storage – the element that is most relevant to the research presented here – the EGEE had a system called the “Storage Resource Manager” (SRM) [44], which provided a replica catalog based upon Logical File Names (LFNs). To access data stored in the catalog, as depicted in Figure 2.8, a user authenticated with his or her Grid certificate and provided the LFN associated with the file they wished to retrieve. The SRM catalog contained a mapping of each LFN to replicas (SURLs) around the network. The user was able to query these SURL locations to gain access to a TURL, or temporary file location. The TURL provided a real and valid endpoint that the user could contact using a known transport protocol (e.g., GridFTP [45] or HTTPS) to download the file.

To access any service on EGEE, users had to identify themselves through Grid (X.509) certificates, which required pass-phrases to unlock, and needed to have short-term proxies delegated in order to authorize tasks. After a proxy was created, it could be delegated to a MyProxy [46] server, where other services could retrieve and subsequently use it. In addition to the creation and delegation of proxies, user credentials were often classified, based upon their Virtual Organization (VO), in the Virtual Organization Membership Service (VOMS) [47]. VOMS allowed grouping of users based upon their security credentials, providing means to partition the user-space and allowing granular authorization. It should be noted that the negotiation steps performed to access a file always involved a user’s (private, personal, and never to be shared) certificate to identity him/her on the network. At no time were files provided or exposed to non-authenticated parties.

EGEE provided researchers with access to computing resources on demand, from anywhere in the world and at any time of day, provided they had the appropriate credentials and allocations. Ease of access and the ability to analyze a larger amount of data within short timescales attracted participation from a wide range of scientific disciplines. When the EGEE-III project was finally completed in April 2010, it boasted 13,000 users, with $\approx$13 million jobs per month, running on a worldwide multi-institutional and transnational network comprised of over 300 computing centers.

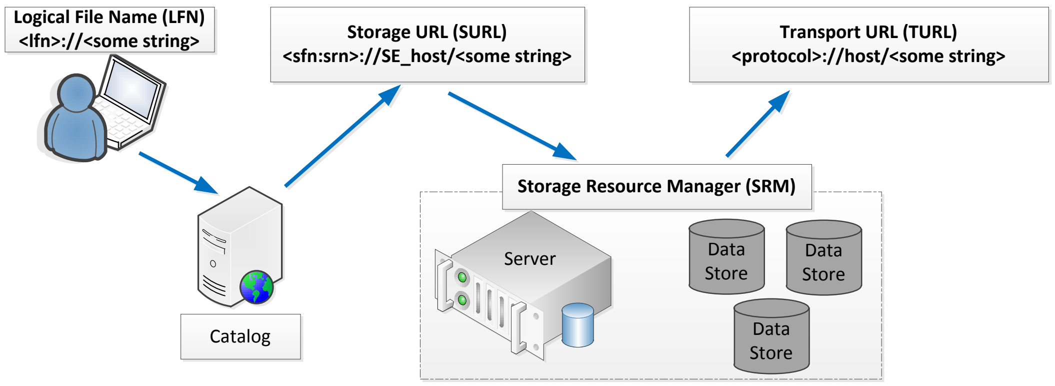


Figure 2.8: EGEE Data Access — shows a simplified view of data access within the EGEE Service Grid computing platform.

It is useful to compare the EGEE achievements with those of the Desktop Grids presented in §2.3.3. Even though the total CPU cores of EGEE rose from 76,000 in 2008 to almost 200,000 at the end of the project, this was still dwarfed by the combined power of BOINC in terms of sheer computational muscle. However, what differentiates the EGEE from Desktop Grids and makes it such a worthwhile investment is the fact that it can run high-performance applications, rather than only the highly scalable "bag of task" simulations that thrive in Desktop Grid environments.

Much of the EGEE infrastructure, tools (notably gLite [48]), and concepts live on in the European Grid Infrastructure (EGI) [11]. National Grid Initiatives (NGI), such as the UK's National Grid Service (NGS), provide the Grid infrastructure and support for scientific computation in their respective countries. The EGI's role is to manage the international collaborations between these NGIs that enable (as was done under EGEE) individual researchers to share and combine computing resources in international collaborative research projects. The EGI is active today and has several related projects that support it, such as the Integrated Sustainable Pan-European Infrastructure for Researchers in Europe (EGI-InSPIRE) project and e-ScienceTalk [49].

Given the vast resources in Desktop Grid environments, and the potential to increase their sizes for relatively small investments, coupled with the high cost of creating a large service Grid, it is very desirable to shift any applications from Service to Desktop Grids if they can function in that environment. By migrating jobs between the platforms, researchers and administrators can help preserve the costly and complex Service Grid infrastructure for the tightly coupled MPI applications it is designed to support, thus reducing overall costs and increasing usability.

2.4.2 Enabling Desktop Grids for e-Science (EDGeS)

One of the obstacles in transitioning Service Grid jobs that could run on Desktop Grids to the new environment is user uptake. Scientists invest a large amount of effort in getting their applications running on any infrastructure, especially one as complex as a Service Grid. Any move to transfer these jobs to a new and vastly different platform would come at a high cost, with perhaps limited direct benefit – unless, of course, the project had substantial and sustaining data-analysis requirements like Einstein@Home and SETI@Home. Naturally, the Service Grid would become less taxed and be able to accommodate more tightly coupled jobs if more applications were migrated. However, a scientist who has invested time in transferring his or her job to a Desktop Grid does not necessarily directly see this benefit. What the scientists could potentially reap as the transfer reward would be the increased CPU availability that can come in a Desktop Grid infrastructure.

Unfortunately, the promise of limitless CPUs in Desktop Grids, and in particular volunteer computing environments, comes at a cost. Technical hurdles aside, before any application can successfully run on a volunteer Desktop Grid, donating users must be convinced to commit their resources to that particular application. For an individual scientist, or a small- to medium-sized research group, this might prove difficult. For example, in the current BOINC community, the top five projects (all of them large and well-funded) account for a disproportional 50% of all donated resources. Beyond the difficulty in building and maintaining a user base, transitioning applications to a Desktop Grid requires a substantial hosting infrastructure to serve input files, retrieve output, and distribute working units. For larger projects, the benefits can far outweigh the costs; however, for the majority of the 13,000 users involved in the EGEE, deploying a Desktop Grid infrastructure might prove unattainable.

The Enabling Desktop Grids for e-Science (EDGeS) [50, 51] project sought to bridge this gap by providing tools and infrastructure to transfer jobs automatically from Service to Desktop Grids, with only minor modifications to the job code [52], and without the need for end-users to set up and maintain complicated infrastructure. EDGeS began in 2008 with the goals of researching and developing "bridging technology" to move jobs from EGEE to BOINC/XtremWeb, and *vice versa* (see Appendix B for more information). To achieve these lofty goals, a number of obstacles had to be overcome. For example, most applications written for the EGEE use the Message Passing Interface (MPI) for coordinating work between processors, something not available on Desktop Grid clients. Typically, EGEE jobs also rely upon local disk access or the EGEE's internal SRM system (see Figure 2.8) to retrieve input files and store output. Moreover, job management and queuing systems in the two environments are vastly different; therefore, there existed a need either to integrate the systems or somehow develop adapters that are able to transition between the two disparate environments.

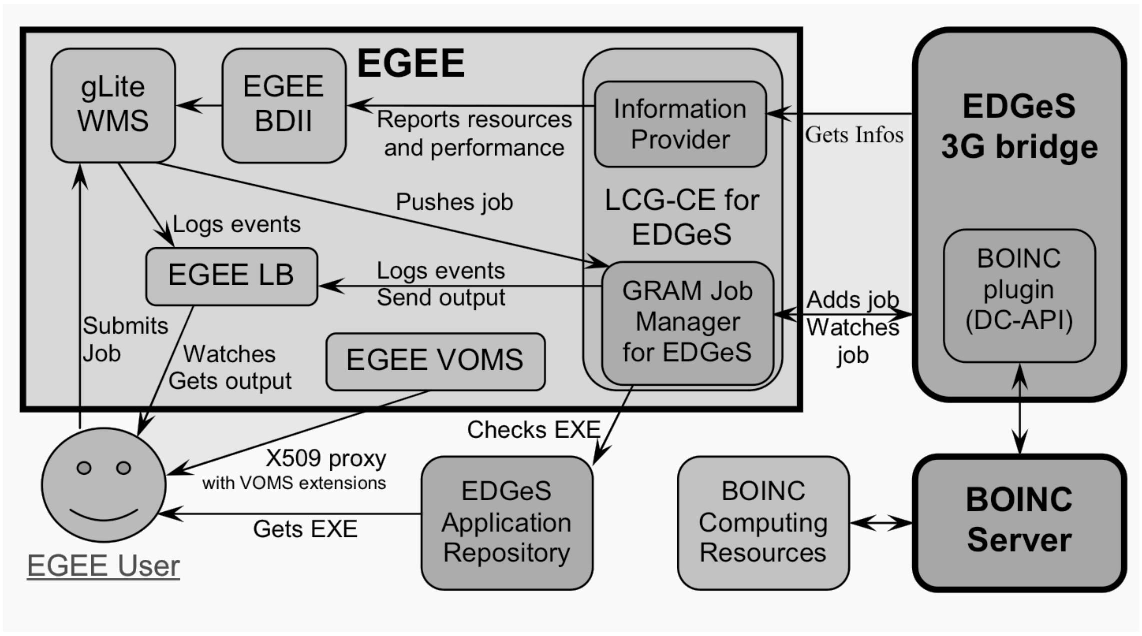


Figure 2.9: Job Migration from EGEE to BOINC — shows how job migration takes place within the EDGeS architecture.

Figure 2.9 gives an overview of how an EGEE job was transferred to BOINC in EDGeS, and could still be transferred using the follow-on EDGI infrastructure discussed in Chapter 6. To begin, an EGEE user logs onto an EGEE resource and submits a normal job from the command line (described by the Job Description Language (JDL)) to the Workload Management System (WMS), which acts as a meta-scheduler. Once WMS receives the job, it queries for information about available resources and finds, among the available resources, the modified computing element of EDGeS. The user can also directly suggest the EDGeS job manager to handle the job. Either way, the WMS system is then able to submit the job to the EDGeS GRAM [53] job manager. The job manager retrieves the executable from the EDGeS application repository, where validated executables are stored, and sends it to the Generic Grid to Grid (3G) [54] bridge. The 3G bridge will translate the job into (in this case BOINC) work units and add them to the queue. Once in the Desktop Grid queue, they are retrieved by clients through a *pull* mechanism, are executed, and any resulting output is returned. For more in-depth detail on how EDGeS developed these solutions – including its core component, the 3G bridge, which moves jobs from Service to Desktop Grids (and *vice versa*) – see the work by Caillat [55] and Kacsuk [56] and colleagues.

Beyond the validation and movement of an application, its metadata, executable, and parameter files, input and output data must also be managed when migrating jobs between Service to Desktop Grids. Notice that Figure 2.9 makes no mention of the input data and how they are transferred to worker nodes. For smaller file sizes and low throughput applications, this could be achieved by hosting input files on the EDGeS BOINC server. However, this solution quickly becomes a bottleneck for large input files or popular applications where thousands of clients might connect.

One key component to EDGeS was the ability to satisfactorily handle the data requirements that arise during job migration [57]. In a traditional Service Grid or high-performance cluster environment, large data input and output files can normally be stored on centralized and well-connected mass storage systems, thereby facilitating the easy access and sharing of data between individual compute nodes. When a job is moved from a Service Grid to a Desktop Grid environment, these systems are unlikely to be either available or accessible to Desktop Grid worker nodes. The lack of data availability produces a need for a new management infrastructure that can get the required input data for a given job and propagate it to each individual client machine that will be performing calculations.

Knowing the need for a new data management paradigm to integrate Service and Desktop Grids, the EDGeS project devoted one of its three joint research activities (JRAs) to this task. JRA3 was tasked with the research and development of a data management solution that could support the transfer of data to and from these two disparate environments. The data access and data-handling solutions developed in EDGeS are extremely relevant to the work presented here because much of the research and software supporting the hypothesis presented in Chapter 1 was developed in and funded by EDGeS. I served as the work-package leader for JRA3. Throughout this dissertation many of the data management topics, research ideas, and subsequent solutions were shaped by the requirements not only of data distribution *within* Desktop Grids but also of *data transition* from Service Grid environments to Desktop Grid computing platforms.

2.4.3 Data Management Obstacles

For EDGeS and projects like EDGeS, data availability could be achieved easily if the Service Grid layer (e.g., EGEE's SRM) was exposed directly to Desktop Grid participants, which had all the necessary libraries and credentials to access and retrieve the needed information. However, this would both require modifications to the Desktop Grid software (i.e., BOINC and XtremWeb) and expose Service Grid file systems to public networks and insecure computers. Although such an approach would closely mimic the current functionality found in most Desktop Grid projects, where data are distributed to all participants from a central machine or through a set of known and static mirrors, it is not feasible or desirable given its security implications and unrealistic software dependencies.

Beyond security implications, Service Grid systems might not technically be able to service the increased requirements that come with deploying jobs to a Desktop Grid. In most cluster environments, data are stored locally or copied to a local file system prior to job execution, allowing many processors fast and easy access to a shared disk. Once a job is moved to a Desktop Grid, each peer wishing to perform work must download an individual copy of the data to be analyzed. For example, a parameter sweep application (see Sequence Correlations in Table 2.4 below) needing to run 100,000 jobs using a 100 MB input file would require 10 Petabytes of network traffic from the Service to the Desktop Grid. Even in less dramatic cases, the additional load of serving every input file to each individual node over a Wide Area Network (WAN) can quickly lead to a very large drain on network bandwidth, in addition to potentially taxing I/O systems. This is especially true for distributed storage systems such as EGEE's SRM, where individual replica locations, depending on their specifications, might not be able to handle the increased loads.

EDGeS required a system that could adapt to varying input-file sizes and replication factors without unduly stressing or exposing EGEE resources. EGEE security infrastructure and policies prevent access to local files from foreign and untrusted hosts, meaning that they somehow have to be transferred to an intermediary location for distribution. Anonymous access is generally not an issue for

BOINC (and less so XtremWeb) projects because they are able to have dedicated and network-isolated data servers. However, the EGEE SRM strictly required X.509 certificate authentication to access data; volunteer nodes would not have access to the needed keys and software libraries to make use of them. Complex solutions for proxy delegation could be developed, but again, these would require major modification to the BOINC and XtremWeb client and server codes (a very undesirable solution that could adversely impact volunteer participation), quickly making such solutions problematic, both technically and politically. Even if these steps were feasible, any solutions would be particular to the individual Service and Desktop Grid systems and would not provide a generic and flexible way of publishing data that could be used by other scientific projects with different Service Grid infrastructures.

2.4.3.1 SG $\rightarrow$ DG Use Cases

Table 2.4 shows the data parameters for many of the applications that were involved in EDGeS.[3] It should be apparent that in several cases the data sizes and per-job throughput requirements far exceed those used by the popular and ongoing volunteer projects seen in §2.3.3. Data volume and access requirements make it impractical to facilitate the bridging and hosting of these applications with a traditional centralized model. Additional roadblocks to easy integration occur when one considers that EDGeS facilitates the bridging of an arbitrary number of applications, each with different application sizes and user-bases. It is therefore unlikely that either the EDGeS Desktop Grid server or the end-user (i.e., the scientist running the job) will have sufficient storage and server capacity to host all the work units and input data. This is especially true for transient applications, where the work isn't sustaining like Einstein@Home, but rather is a short-lived project (i.e., a short research grant or Ph.D. student validating his or her results) that doesn't have the resources and time to construct a mirrored server array to host input files.

The EDGeS 3G Bridge, introduced in §2.4.2, is meant to be a generic platform to transfer jobs from Service to Desktop Grids. As part of this role, and to deliver a functioning system, the infrastructure must provide a way to host or distribute input data files. Since EGEE is a decentralized platform, or a federation of institutions, there is no "centralized" place to host all these files in a public-facing fashion,

[3] For more information and applications, see: http://www.edges-grid.eu/web/edges/49

Table 2.4: EDGeS Applications' Data Management Needs

Name	Files/WU	File Size[gh]	Output[g]	Exec Time	Tasks/Day[g]
ISDEP	6[a]	2	1	30 min.	50,000
pLINK	2	380	0.5		
ViSAGE	2	2	1	1 min.	10,000
Desktop Grid Pattern Finder (DGPF)	2	0.5	0.5	5 min.	3,100
Distributed Audio Retrieval using Triana (DART)	6[b]	52	0.01	2 min.	1,000
itemgrid	1	<1	<1	60 min.	1,000
E-Marketplace Model Integrated with Logistics (EMMIL)	1[c]	1	10	10 min.	1,000
X-ray	0.1[d]	0.2	1	20 min.	10,000
VisIVO	4[e]	1000	50	30 min.	2,000
GT4Tray	1 – 1000	1 – 1000	1 – 1000	1 – 240 min.	200
Multiscale Image and Video Processing	5[f]	1	1	20 min.	1024
Sequence Correlations	1	100	50	180 min.	100,000
MOPAC	3	0.1	0.5	4 min.	100,000

[a] Does not include pre-stage files of 80 MB for each new host.
[b] Does not include pre-stage files of 312 MB for each new host.
[c] Does not include pre-stage files of 80 MB for each new host.
[d] Does not include pre-stage files of 2 MB for each new host.
[e] Does not include pre-stage files of 15 MB for each new host.
[f] Does not include pre-stage files of 150 MB for each new host.
[g] Estimation is of the upper bounds of the number of tasks that would be available each day for processing.
[h] Sizes represented in megabytes.

as needed by a Desktop Grid. Arbitrarily choosing a specific site to host all the EGEE$\rightarrow$DG files (such as an EDGeS project server) would quickly prove to be a bottleneck, as well as unsustainable, because it would disappear as the project finishes, and/or as different groups join or leave the EGEE and wish to leverage bridge technology.

It becomes apparent that a system is required that can dynamically restructure itself to accommodate the changing nature of EGEE membership, grants, and infrastructure improvements, as well as provide the measurable Quality of Service (QoS) that is needed to host the input files. A dynamic self-load-balancing system is needed, with the ability to reconfigure, add new servers, and change relative loads as the applications and needs evolve. The characteristics described here closely mimic those found in “peer-to-peer” networks, making them worth exploring as a possible research avenue.

2.5 Peer-to-Peer Networking

The term “peer-to-peer” (P2P) has generally been categorized as the ability to use resources on the “edges of the Internet” [58–60]. The “edges of the Internet” refer to those computers, systems, and devices that are not part of the integral backbone of the Internet, but rather sit on the periphery, such as home PCs, mobile devices, and other consumer-grade and widespread electronics. One of the promises of P2P computing is to leverage these heterogeneous and distributed resources to perform complex and useful tasks, where the system aggregate provides non-trivial levels of service. Following this loose definition, volunteer computing projects (e.g., BOINC) can be described as peer-to-peer even though they are centrally managed and communication is client/server.

More narrowly, *peer-to-peer networking* refers to a system where the participants interact on similar terms and can participate, more or less, as equals. In P2P networking, the focus is on providing the environment and the tools to connect many heterogeneous and dynamic resources together, in interactions and roles that are not restricted to traditional client/server models. Thus P2P networking is about how to enable what can be complex and highly dynamic relationships among a variety of software or hardware resources. P2P then becomes a term that describes a general problem domain and a set of solutions on how to address interconnectivity issues in a highly dynamic and unreliable environment. In the data management domain, P2P has proved highly successful for distributing files among millions of users, and “peer-to-peer” is used to describe both the network topology and a social organization scheme in which participants operate as servers and also as clients, often concurrently, depending on the circumstances, network requirements, and user demand.

When constructing a peer-to-peer network, it is important to keep in mind the great heterogeneity of the devices. Unlike the main Internet routers, systems, and servers that comprise the backbone of the Internet and provide much of the infrastructure that we rely on, devices at the “edge of the network” vary greatly in their capabilities. For example, a consumer laptop might have significant storage abilities and processor speeds yet lack any GPS function, whereas a mobile phone might have GPS and sufficient storage yet be constrained by minimal memory and battery issues. Also important is the realization that in a P2P network, the home computers, mobile devices, and other systems that comprise the system are extremely volatile. At any moment they can be switched on or off, thereby changing the network topology and thus requiring a system that can easily adapt to a great number of failures and a constant need for reconfiguration.

2.5.1 The Explosion of P2P File-Sharing

Many software systems have been developed since the advent of the consumer Internet, but perhaps none has been as controversial as peer-to-peer file-sharing. P2P systems and their uses have been central in the debate over how information and data are shared on the Internet (and how to prevent it). When discussing the history and developments in distributed file-sharing, it is useful to explore the (somewhat controversial) history and evolution of peer-to-peer data networks, which has led to much of the technical innovation in the distributed computing field.

Napster [61] was perhaps the first famous use of peer-to-peer file-sharing, attracting millions of users. It provided a user-friendly and easy way for people to use their PCs to connect to other peoples’ PCs and exchange files. The majority of the files transferred on Napster were MP3 encoded music files, the majority of them copyrighted. To orchestrate the millions of connected systems, Napster servers acted as metadata caches for locating information, with the end-data (i.e., music files) being stored on the participants’ personal home computers. Once a requested file was located, the requestor would directly download it from the hosting machine. Participants could both receive and send files, and even share partial copies of files with one another to create whole sets. Therefore, in the Napster network, the location of information was completely centralized on their servers, but the distribution was “peer-to-peer.” This model is often referred to as a “brokered” approach, and it was extremely successful, allowing the Napster network to successfully scale to a peak of 25 million users and 80 million files. Since much of the network traffic on Napster was copyrighted music and its use was widespread,[4] lawsuits from the music industry quickly ensued. Through a combination of legal actions against Napster and a fear of stiff fines directed toward users, Napster's popularity fell, to be replaced by several other peer-to-peer systems such as Limewire, eMule, Kazaa, and Gnuttela.

Armed with the knowledge that a large part of Napster's demise had been due to its centralized architecture – which, although technically sufficient, provided a single entity that could fail or be targeted for "shutdown" – Napster's successors investigated new distribution mechanisms. The remainder of this section discusses the evolution of peer-to-peer file-sharing and the new technologies that began to be developed in the years following Napster.

2.5.2 Peer-to-Peer File Distribution Taxonomy

Practically every website on the World Wide Web (WWW) uses a centralized model to host its webpages, files, and other content. In a typical Internet hosting environment, a Web server (or series of load balancing Web servers) is used to satisfy requests for multiple clients. Such a straightforward and centralized approach works quite well if all data can be aggregated to one location and the servers can successfully scale to meet demand. Most large companies, especially those involved in e-Commerce, such as Apple and Amazon, ensure that they have sufficient servers to satisfy anticipated peak demand, although this requires "over-building" capacity to meet transient and infrequent periods of peak demand. Amazon's over-building of capacity to meet peak demand led to its release of its Cloud computing infrastructure, as detailed in §2.2, to sell the extra cycles when they aren't being used. For companies and websites that lack additional capacity, when something unexpected becomes popular or in high demand, such as a particular news story or website, an effective denial-of-service attack can be launched when bottlenecks are reached, grinding the targeted website or service to a halt.

Despite advances in P2P networks, attempts to provide distributed website caching [62], and the introduction of Cloud computing, centralized distribution and control have remained the pivotal means of sharing most content, and for

[4]Other file-sharing networks had existed before, but they had not been nearly as successful and easy to use, and therefore not as threatening as Napster to copyright holders.

good reason. Cloud computing requires a migration cost for existing systems and often a “buy-in” to a particular Cloud provider’s system, as well as a direct sustaining cost that must be paid to host the services. Peer-to-peer networks have a reputation for being unreliable and the plaything of Internet pirates, and have only recently begun to shake this stigma. Additionally, many P2P systems provide only “best guess” estimates of availability and limited quality of service, making them inherently unreliable for real-time interactions such as website hosting. File-sharing, however, is not as interactive as webpage requests, with a natural latency built into any system where large amounts of data are being transferred. For file-sharing, the scalability and distributed storage benefits of P2P often outweigh any potential network latency or other quality-of-service (QoS) metrics that might be sacrificed; this makes it an attractive technology.

2.5.3 Flat Networks

As Napster began to face legal trouble in 2000, other peer-to-peer technologies came to the forefront as new ways to share files, without relying on the centralized system that would prove to be Napster’s downfall. The first of this new breed of P2P software protocols eliminated the centralized tracker system found in Napster and instead found metadata by querying peer nodes directly in a rudimentary method called “network flooding.” In network flooding, messages are broadcast throughout the network in a very egalitarian approach for a certain number of hops (routing depth), until they at last expire or have reached all nodes in the network.

Flat networks are advantageous in the sense that they give very accurate and up-to-date search information by going directly to the source (i.e., end machines that contain the data) to query for availability. However, they suffer from a significant disadvantage due to the high message overheads incurred by network flooding, which becomes especially pronounced as network size increases.

2.5.3.1 Gnutella (Early Versions)

The Gnutella software was released under the GNU General Public License (GPL) [63], making it available to freely use, reuse, and modify. This is in stark contrast to Napster, which was a proprietary and closed-source software program distributed and developed by a private company. Gnutella’s open-source availability, as well as timing, made it highly popular. The ability to change and reuse the Gnutella software and protocol also allowed it to be more dynamic than Napster was, evolving over time from a community of interest and adapting to new distribution needs and network advancements. The first version of Gnutella was a completely unstructured network that used network flooding to locate information. Despite its advantages of being decentralized and dynamic on large volatile networks, such as the void left by Napster, network flooding proved highly inefficient. As more messages flooded the network, speeds slowed to a crawl. Not only are connection operations expensive in terms or resource utilization, but querying an entire network of millions with metadata requests can become a huge network load, trumping that of the actual data being sent.

The limitations of deep network flooding became apparent to Gnutella developers, and by version 0.4 of the protocol, they sought to gain scalability by limiting search depth to seven hops, and having peers connected only to five other nodes. The new flooding algorithm yielded a maximum potential network flood of 5^7 routed messages for each query – substantial, yet only a small fraction of the network. This still proved inefficient and unscalable in such a large network, often led to inaccurate results (since only a small partition of the network was being searched), and had a reputation for still rendering the network unusable by low-bandwidth machines because too much traffic was being consumed by metadata queries.

Gnutella's failure to scale on a flat network should not be seen as an indication that flat networks did not have their place in peer-to-peer environments. For small-sized networks such as LANs or mobile *ad hoc* networks, network flooding approaches can be extremely effective. Apple's Bonjour protocol is an example of this and is Hewlett Packard's direct jet printer software. The User Datagram Protocol (UDP) and other network protocols that are used every day by our computers also use network flooding to locate resources and send messages.

After Gnutella proved flat networks to be unfeasible for Internet-scale connectivity levels, other metadata searching and management solutions began to be explored, namely, centralized/decentralized, distributed hash tables, and "super peer" topologies.

2.5.4 Centralized/Decentralized

As opposed to a completely centralized solution, a centralized/decentralized approach provides the reliability of a centralized server with the scalability and task-sharing of a decentralized network. A centralized/decentralized model can be very successful in providing up-to-date, accurate, and efficient metadata management, coupled with a load-balanced distributed mechanism. However, it also has the disadvantage of a centralized entity that can become a bottleneck or otherwise fail. Centralized/decentralized models are often referred to as a "brokered" approach, because a centralized entity "brokers" the interactions among a large group of network participants and acts as a central organizer for communications.

2.5.4.1 Napster

Napster [64], the first highly successful use of peer-to-peer file-sharing, was launched in 1999 and finally closed down in July 2001. It was run by a private company that hosted all metadata about the network and its content at the company's centralized servers. The vast majority of all data on the Napster network consisted of MP3-encoded music files. To participate in the Napster network, clients would connect to the Napster server, where they would register the files they wished to share and could also initiate search requests. Once queries matches were made, a user could initiate download requests directly with other network participants without routing any "data" traffic through Napster's servers. The model was extremely successful and scaled to a peak of 80 million hosts before Napster was finally sued over copyright concerns and its user-base migrated to alternative networks.

2.5.4.2 BitTorrent

BitTorrent [65] was created in mid-2001, and remains a popular file distribution protocol to date. Like its predecessor, Napster, BitTorrent follows a centralized/decentralized approach and is extremely successful and efficient in distributing files. All file metadata are stored in one location,[5] a centralized tracker that has an index of each node on the network that contains parts of the targeted file. Un-

[5]There are alternative trackers for BitTorrent which allow the tracking files to be distributed, as well as distributed tracker indexes. However, the default and most widely used implementation of the BitTorrent tracker is centralized.

like Napster, locating the tracker (and therefore the data) is out of the bounds of the BitTorrent protocol and depends on other metadata engines, websites, or indexes. BitTorrent is most beneficial when distributing frequently used and sizable data [66], such as Linux distributions and, more controversially, copyrighted movies and music.

Similar to the legal battles that faced Napster, the indexes that share information about where to find BitTorrent trackers are often targeted for shutdown by those who believe their copyrights are being violated. However, what differentiates BitTorrent from Napster is that each tracker contains only the information about one file. In BitTorrent there is no central metadata repository, and no tracker index is central to the existence of BitTorrent. Unlike most P2P networks, BitTorrent does not create one large network; rather, it creates many small networks, each of which focuses on the efficient distribution of a single file. BitTorrent has survived the legals battles that disrupted Napster partially because its software and protocol are generic and can be used for distributing any type of file (and often do so). Also, the company that created BitTorrent does not host the entire network's metadata, which would inevitably include links to copyrighted material. The result for BitTorrent is a strong case for plausible deniability that the software is being used illegally, weakening any legal basis for shutting down BitTorrent and identifying the protocol as a distributor of illegal software products.

To obtain information about a file to download, a peer must download a corresponding .torrent file. This file contains the file's length, name and hashing information, and the url of a tracker, which keeps a global registry of all the peers sharing the file. Trackers help peers establish connections between each other by responding to a user's file request with a partial list of the peers having parts, or chunks, of the file. A tracker does not participate in the actual file distribution; each peer decides locally which data to download based on data collected from its neighbors. Therefore, each peer is responsible for maximizing its own download rate.

One notable difference between BitTorrent and other P2P software is that BitTorrent does not incorporate peer and file discovery algorithms. Its main focus is on optimizing the distribution of files. This is done by enabling multiple download sources through the use of file partitioning, tracking, and file-swarming techniques. One feature in BitTorrent that has diverged from many other P2P software products is its enforcement of collaboration between users by a mandatory fair use doctrine called "tit-for-tat." In tit-for-tat, network participants are (generally) able to receive file segments only if they are also providing them, a policy that enforces fairness in the network and helps maintain network parity.

Although BitTorrent is a very effective software system for distributing large, frequently used files, its tit-for-tat policy and lack of a centralized tracking mechanism make it unusable for BOINC data distribution. However, many of the concepts employed by BitTorrent, such a file-swarming, two-stage MD5 hashing, and centralized tracking (i.e., brokered architecture), are very applicable and useful for Desktop Grid data management.

2.5.4.3 BlobSeer

BlobSeer [67] is a data management system designed to support high-throughput data-intensive applications over a wide-area-network (WAN). To do this, BlobSeer employs a Napster-like topology, where there is a “provider manager” that keeps information about storage space and schedules placement of new data items. Data is stored in Binary Large Objects (BLOBs), which act as containers for smaller application data. This has the benefit of limiting the amount of unique identifiers on the network, and lowering metadata management costs. Full metadata and storage information is distributed across the network for scalability and redundancy. The target community for BlobSeer is MapReduce [68] applications, and the infrastructure can be accessed and used through a custom application programming interface (API).

2.5.5 Distributed Hash Tables

Distributed hash tables (DHTs) rely on structured lookup algorithms to locate information. As the name suggests, a distributed hash table acts much like a local hash lookup, with unique keys that map to values, although a DHT operates in a distributed manner. There are many examples of DHT networks and the underlying protocols and routing algorithms that make them possible. Tapestry [69], Pastry [70], CAN [71], and Chord [72] are some of the most well known. They all share the same concept — that individual pieces of information, referenced by unique keys, are stored in structured locations that can be identified and found. To locate information, a search is sent to a node on the network. That node will then see if it is responsible for that key space, and if so, respond with the relevant data. If the node is not responsible for that key space, it will look in its neighbor table (a list of other nodes it can connect to) to see who is "closest" to the key space in question, and route the message to them. This sequence continues until the information is found, or a node cannot find a neighbor with a closer mapping, in which case the key/value pair does not exist on the network.

A simple example of a distributed-hash-table-style distribution would be to take all the words in the English language and assign them to 26 different nodes, with each being responsible for all words that start with its assigned letter. Each node has a certain number of neighbors it keeps track of (network "degree"), which we will set to two. For simplicity's sake, let each node have only one neighbor, and let it be the next letter in the alphabet. If a node at position A wanted to search for a word starting with the letter D, it would have to traverse three hops before it arrived at its final destination: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D. In this simplistic scenario, the maximum route length a query would have to traverse before locating an answer would be N, where N is the size of the network.

There is a direct relationship between *routing length* and *degree*. The greater the degree, the less routing length needed, and *vice versa*. A network with N degrees, where N is equal to the size of the network, would be a terribly inefficient and very "flat" network with extremely high maintenance costs when nodes enter and leave the network, since the entire overlay would have to be rebuilt on every entry or exit. Conversely, a network with a very low degree would require potentially extremely long routing lengths to find information, leading to excessive message overheads and network traffic.

DHTs retain lookup tables that help map a particular request to the appropriate node, and they generally have a list of neighbors that help requests to find responsible nodes in a structured manner. When using a DHT-type system, users are normally guaranteed a certain efficiency for any given lookup, with a set number of hops relative to the size of the network. It is also generally accepted that in DHT networks, if a given piece of information is on the network, it can be located. These types of systems are extremely powerful when searching for known and structured data.

The most common choice for DHTs is to use a symmetrical degree/routing length of O(log n). Degree/route length is not optimal in terms of degree/route length tradeoff, although such topologies typically allow more flexibility in choice of neighbors. Many DHTs use that flexibility to pick neighbors that are close in terms of latency in the physical underlying network. Table 2.5 gives an overview of several routing-length to degree comparisons in DHTs, which are governed by the degree/diameter tradeoff that is fundamental in graph theory [73].

Table 2.5: DHT Efficiency — shows Degree vs. Routing Length

Degree	**Route length**
O(1)	O(*n*)
O(log *n*)	O(log *n* / log(*n* log *n*))
O(log *n*)	O(log *n*)
O(1)	O(log *n*)
O($\sqrt{n}$)	O(1)

2.5.5.1 OceanStore

OceanStore [74] is a global, distributed, Internet-based storage infrastructure. It consists of cooperating servers that work as both server and client. The data are split up into fragments that are stored redundantly on the servers. For search, OceanStore provides the Tapestry subsystem, and updates are performed by using Byzantine consensus protocol. However, this adds an unnecessary overhead since file search is not a requisite for BOINC and supporting replication implies the use of a distributed locking service, which incurs further performance penalties.

2.5.5.2 Freenet

Freenet [75] is fully distributed but employs a heuristic key-based routing in which each file is associated with a key, and files with similar keys tend to cluster on a similar set of nodes. Queries are likely to be routed through the network to such a cluster without needing to visit many peers. Freenet is concerned with building a censorship-resistant network for information and not necessarily a distributed file system. Although Freenet uses keys and routing similar to a DHT, Freenet isn't strictly a DHT system and does not guarantee that data will be found. The high focus on anonymity and the inability to guarantee data availability make Freenet a poor choice for BOINC data distribution.

2.5.6 Decentralized Super-Peer Topologies

Whereas flat networks are the purest form of an unstructured network, a super-peer topology takes the flexibility of a flat network and enhances it by promoting more capable nodes to greater responsibility. In these types of systems, the more capable nodes, or “super peers,” take the responsibility of message relaying and forming the base network topology. Although the method of distributing the queries in this type of system can have a larger overhead, it also allows for a greater flexibility than does a DHT-style system, by providing for custom and fuzzy matching of data to queries.

A super-peer network is one where all entities on the network are not created equal, as they are in a flat network. Some of the peers act as metadata cashers, and they keep track of a certain subset of the network and what is located on it. This allows for much faster searching and retrieval of information. Instead of an individual node being required to search through all other members of the network, the super peers aggregate and relay the requests to other super peers that contain metadata stores. One of the side effects of having a super-peer network is that when one of the super peers goes offline, the network could become partitioned and/or metadata could be lost. This risk can easily be eliminated by overlapping the metadata that each super peer has and by a having edge nodes connect to more than one super peer (e.g., three in Gnutella). The amount of metadata replication between the super-peer layers dictates how much network volatility can be tolerated before failures begin to occur: more overlap reduces failures, but increases network load and synchronization.

The decentralized super-peer approach was quickly adopted by the successors of Napster as the preferred method of constructing peer-to-peer networks. The most popular software programs that developed and used super-peer technologies were Kazaa, its clones[6] LimeWire and Morpheus, and Gnutella.

2.5.6.1 Gnutella (Version 0.6 and Up)

Gnutella [76] built upon its previous work to construct a decentralized file-sharing system, seeking ways to lower message traffic overhead and make it scale to millions of users. This was done by abandoning a fully distributed approach and adopting the notions of “ultrapeers” and “leaves.” Ultrapeers became responsible for message relaying and were typically better computers on higher-speed connections not located behind firewalls. As such, they served as routers for requests

[6]Kazaa developed the FastTrack network which used super nodes to improve scalability (and later became the basis of Skype). However, the Kazaa client software was known to install malware and adware on users’ machines. LimeWire and Morpheus were both developed as malware-free alternatives to Kazaa that piggybacked on the FastTrack network.

and responses for leaf nodes (i.e., normal peers). Through the use of “ultrapeers” (aka super peers), a network overlay was created out of a small subset of the entire network. Because the overlay network was smaller and generally comprised of higher-performance and less volatile machines, it could more easily scale and route messages.

In the previous versions of Gnutella (see §2.5.3.1), each node was connected to approximately five peers, and search queries were sent out onto the network with a maximum of seven *hops*, yielding a maximum of 5^7 recipients for a given request and a large amount of network traffic. Within the new ultrapeer overlay, each leaf node is typically connected to three ultrapeers and each ultrapeer is connected to at least 32 others. With this higher outdegree (and larger number of potential recipients), the maximum number of hops a query can travel was lowered to four. This modification dramatically reduced the message overhead required for locating information and maintaining network connectivity, enabling Gnutella to scale to millions of participants.

2.5.6.2 Kazaa

Kazaa [77, 78] is similar to Gnutella, although it extends upon this by exploiting peer heterogeneity and organizing the peers into two classes, Super Nodes (SNs) and Ordinary Nodes (ONs). SNs are generally more powerful in terms of connectivity, bandwidth, and processing power, and are not behind NAT systems. In order to bypass firewall and NAT systems, Kazaa uses dynamic port numbers along with a hierarchical design where a node can act as a relay between two other nodes. Like Gnutella, Kazaa’s file discovery mechanism creates unnecessary traffic and its Super Node architecture applied to data distribution on BOINC could generate an unacceptable level of network traffic while relaying requests.

2.5.7 Tightly Coupled LAN-Oriented File-Sharing Systems

There are a number of highly effective P2P file systems that were built for high-performance reads and writes. These systems typically operate in an environment where network speeds are high and latency is low. Although not applicable to the BOINC environment, which is a widely distributed WAN with varying speeds and network connectivity, it is worth mentioning these variants of P2P file-sharing systems, in order to provide a comphrensive overview of the technologies and breadth of the P2P file-sharing realm.

2.5.7.1 Google File System (GFS)

Google File System (GFS) [79] (latest incarnation dubbed “Colossus”) is a distributed storage solution that scales in performance and capacity while being resilient to hardware failures. GFS was designed to operate in a trusted environment where the application is the main influence of usage patterns. The GFS typical file size was expected to be in the order of gigabytes and the application workload to consist of large continuous reads and writes.

2.5.7.2 FreeLoader

FreeLoader [80] aggregates unused desktop storage space and I/O bandwidth into a shared cache/scratch space for hosting large, immutable datasets and exploiting data access locality. It is designed for large scientific results (outputs of simulations). FreeLoader focuses on aggregating space and bandwidth in a corporate LAN setting. It adopts a certain degree of centralized control in data placement and replication, for better data access performance. The overall architecture of FreeLoader shares many similarities to GFS.

2.5.7.3 Farsite

Farsite [81] was a Microsoft Research project from 1999-2005 that aimed to provide users with a persistent non-volatile distributed storage system. Farsite aggregated the space of desktop clients to collaboratively establish a virtual file server, without the need for a central server. It provided a global name space, as well as access control for both public and private files. It was implemented by providing multiple encrypted replicas of each file among a set of client machines, then referencing the files through an hierarchal directory that was maintained by a distributed directory service.

Much of the research into Farsite was figuring out how to ensure high availability and security while building the system from cooperating yet mutually distrusting hosts. Farsite uses the Byzantine agreement protocol to establish trust within an untrusted environment. The project was designed primarily for high-speed LAN networks and as a way to provide real-time access to files. The experiences of the Farsite project are documented in [82].

2.5.7.4 Frangipani

Frangipani [83] was a performance-oriented Distributed Storage System designed in the late 1990s. It was typically used by applications that require a high level of performance. It followed a server-client architecture and was implemented on top of the Petal system, employing Petal's low-level distributed storage services. Like Farsite, it was designed to be used within the bounds of an institution where servers are assumed to be connected by a secure high-bandwidth network. This architecture choice makes it unsuitable for WAN distribution of BOINC files. Furthermore, like OceanStore, Frangipani also implements a distributed locking service, causing a considerable performance drop when servers access the same file.

2.5.7.5 stdchk

stdchk [84] was designed to harvest idle storage space in Desktop Grids and clusters to provide a common space for checkpointing files. One characteristic of stdchk is that it requires high-throughput in its write operations, yet reads are far less important and may never occur. This is because checkpoint files are periodically written to save program state, and only need to be read in the case that the program was halted and needs to be restarted. This design makes stdchk useful for local Desktop Grid or cluster systems, where nodes are interconnected by high-speed networks, yet impractical in a highly distributed network such as volunteer Desktop Grids.

2.6 Bridging the Data Management Gap

The motivation for the research presented here is twofold: first, to develop a solution that can distribute data within Desktop Grids, and second, to provide a data-hosting environment that is able to facilitate the movement of localized Service Grid data into a broader Desktop Grid distribution environment. Fortunately, these goals are complementary, both fundamentally dealing with the distribution of data within a Desktop Grid, with the caveat of additional application use-cases from the Service Grid layer that had not yet been applied in Desktop Grids.

Desktop Grid environments have different requirements from other file-sharing P2P communities because security can become a more complex issue than simply guaranteeing data validity (see §3.4.2). In Desktop Grids, there can be a requisite that only certain amounts of data are shared with an individual peer. Also communities can be reluctant to introduce a system that would have peers directly sharing with one another because this might be perceived to have potential security implications for clients as ports are opened for outside connectivity. It is therefore important not only for data integrity and reliability to be ensured, but also to have available safeguards that can limit peer nodes' exposure to malicious attacks. It is these types of requirements and research questions that has prompted my work to investigate, design, and ultimately create a custom P2P network for data distribution. The goal is to envisage a new type of system that goes beyond basic file-sharing to incorporate the domain-specific needs inherent in the scientific application domain, providing both client and server safeguards and stricter controls for project administrators as to which network participants receive and distribute data.

It is possible to improve the current centralized distribution mechanisms employed by volunteer computing projects, by providing a viable alternative that leverages peer resources and P2P technologies. Several colleagues and I have explored the application of using P2P in volunteer computing as a means to offload the central network needs [85]. There are many ways this could be implemented, ranging from a BitTorrent-style network [65], where data are centrally tracked and all participants share relatively equal loads, to Kazaa-like super-peer networks [86], where select nodes are assigned greater responsibility in the network.

Several of the popular and proven P2P technologies discussed in this chapter, as well as commercial solutions like Amazon's S3 or Google's GFS, could be configured and/or customized and then effectively applied to provide for the data needs of BOINC or XtremWeb, at least as they relate strictly to distribution. However, in the case of commercial products, like the Cloud, there is a direct monetary cost involved, and for P2P systems like BitTorrent, the facility to secure or limit who is able to receive, cache, or propagate different pieces of information is generally limited or nonexistent. These limitations make it difficult to directly adopt an existing technology to the Desktop Grid application domain. However, they do provide a solid technological base and much of the fundamental research that is needed to develop a new technology that conforms to both Desktop Grid data distribution and the Service to Desktop Grid migration requirements.

Applying any P2P infrastructure to scientific computing, and particularly in volunteer computing, can be highly problematic. In such environments, policies and safeguards for scientific data and users' computers become more critical concerns for limiting update rather than any technical feasibility. To be successful, any solution must take into account the requirements of scientific communities, as opposed to focusing on overarching P2P architecture. A balance must be struck between the advantages of facilitating different network topologies and data distribution algorithms and retaining the safety of each participant's computer. Further, each scientific application has different network and data needs; customized solutions would allow for tailoring the network toward individual requirements, despite the disadvantage of increased development effort, complexity, and code maintenance.

When considering using P2P data access technologies in the scientific application domain, two broad challenge areas must be addressed: *social acceptability* and *technological challenges*. Socially, peer-to-peer technologies, especially when used for sharing data, are often viewed with a skeptical eye, having long been associated with widespread file-sharing of copyrighted material.

In addition, there is substantial concern that mixing peer-to-peer with volunteer computing could, in the event of malicious attacks on the network, cause irreparable damage to the volunteers' trust in the network, thereby adversely affecting their willingness to continue donating resources. During the development of this thesis, such concerns were ongoing and took on a very important role during the design process. When researching an appropriate system to build, it is necessary to identify solutions that not only move forward Desktop Grid utilization, but also introduce peer-to-peer networks and P2P file-sharing as both valid and legitimate options for scientific computing.

Within the technical area, security and scalability are the main issues being considered. Scalability for large P2P networks has evolved into two general categories, which were discussed earlier: Distributed Hash Tables (DHTs) and super-peer topologies. Both of these approaches are valid and have their unique advantages and disadvantages depending on the problem one is trying to solve, generally with a tradeoff between speed, accuracy, and flexibility. Finding the correct balance for each individual situation is the important factor. With this in mind, scalability research focuses here on designing an adaptable network that can automatically change its topology to optimally balance network load, an especially useful trait in the case of super-peer technologies, where effective algorithms can help promote an efficient and scalable design.

Security is a much larger issue. Due to the sensitive and vulnerable nature of Desktop Grids, it is critical not only that peer nodes are secure from malicious attacks, but also that data integrity and reliability are ensured. The easiest solution, and the one perhaps the most susceptible to attacks, is a pure P2P network, in which any node is allowed to receive and share information with any other node on the network. This allows for the most network flexibility and client resource usability. However, since in this scenario any node has the capability to promote information, it also has the ability to flood the network with false information. Even though safeguards and hashing can be put in place to mitigate these effects, there is still the potential for malicious network utilization. In a more restricted network, where only "trusted" peers are allowed to act as data cachers and rendezvous nodes, the probability that this will happen is diminished; however, usability and flexibility are reduced as a result.

Beyond EDGeS and its subsequent follow-on project entitled EDGI, the research presented in the rest of this dissertation is focused on providing a solution for the broader scientific community. As shown in §2.3.3, the network bandwidth and server requirements on volunteer computing projects can quickly become large, as the network scales and more people join a project. Other BOINC projects have much larger input files; for example, in Rosetta@Home files are three to four megabytes and have similar computing times to SETI@Home. Network requirements can start to become prohibitive for projects, resulting in scientific problems being often scaled down (e.g., by lowering the resolution of input files or preprocessing data) to fit into the available network space. These requirements and restrictions prevent new forms of applications from migrating to Desktop Grids, like those in Table 2.4. They include applications that have input files in the gigabytes, not the megabytes scale (e.g., analyzing medical imagery and searching large protein databases). In addition, a new push is being made toward running Desktop Grid applications in virtual machines, to facilitate fine-grained control over the application while protecting host machines. However, these efforts are currently being held back by the large data requirements needed to distribute the images. Enabling the applications mentioned here, and many others that are not, to make use of the vast power of Desktop Grids would be a large step toward further enabling computational science, while at the same time providing sustainability paths for applications that have traditionally been re-stricted to supercomputers and other Service Grid infrastructure.

CHAPTER 3

Analysis and Design

Chapter 2 introduced the state-of-the-art in distributed and volunteer computing networks, showing how Desktop Grids would greatly benefit from the application of a robust data management solution. There, many of the top peer-to-peer (P2P) technologies and methodologies were outlined.

This chapter will analyze the requirements for Desktop Grids as they relate to data sharing, and give a check-list for what should be considered when looking to provide new data distribution mechanisms. As will be shown, existing solutions fail to solve some of the fundamental problems that occur in volunteer computing environments. The lack of a direct mapping between existing software and the volunteer computing application domain often is not due to a technical deficiency regarding the actual distribution of files. Rather, it is the insufficient flexibility of existing systems to adapt to the social context of volunteer computing and integrate with legacy applications. Legacy application integration is a very important issue when attempting to apply a new technology to an existing network, especially one comprised of hundreds of thousands of active “volunteer” participants.

In addition to domain requirements, there are a number of scalability and security issues that arise when building a P2P network for volunteer computing. Security in these systems goes beyond the traditional notion of simply ensuring authentication or file integrity. Scalability not only needs to take into account network bandwidth, but also the potential sizes of data as they relate to job throughput and its distribution over time.

Volunteer computing communities can also be reluctant or hostile towards introducing a system that would have peers directly sharing with one another, as it might have the potential (or, almost more important, perceived potential) to have security implications for clients. A true P2P system requires participation and sharing from many nodes, generally necessitating ports to be opened for outside connectivity and untrusted peers to be given network responsibility. Since the success of volunteer Desktop Grids depends on an active and pleased community of donors, the importance of guaranteeing social acceptance from volunteers is of upmost importance. It is therefore imperative that not only data integrity and reliability be ensured by any system to be deployed in this application domain, but also to have available safeguards that can limit peer nodes' exposure to malicious attacks and provide assurances to the community.

It is with regard to these security and scalability requirements, the inherent legacy integration issues, and the social context of the target application domain, that this chapter analyzes and designs a suitable system for Desktop Grid data distribution.

3.1 Methodology

The approach used in this chapter to analyze the problem space and develop a design is first to look at the core user requirements (see §3.4) needed to develop a successful solution that has the potential to be adopted by the volunteer computing community. After identifying a core set of principles that must be upheld, current technologies are analyzed for their mapping to these requirements (see §3.5). Concurrent to exploring existing middleware, the useful mechanisms currently employed, as well as the needed adaptations to create a useable system are noted. These lessons are then applied to the development of a new network paradigm (see §3.6). The resulting network infrastructure and design is then validated through simulations using real-world metrics from the volunteer computing community (see §4.5). This infrastructure is then improved through an iterative simulation process and is used as the basis for both the protocol developed in Chapter 4 and the concrete network implementation described in Chapter 5.

It is therefore with a requirements-gathering process, an analysis of the state-of-the-art, and an exploration into new solutions which are then verified through real world metrics, that this chapter designs a new P2P network for data distribution.

3.2 Problem Scope and Analysis

As volunteer computing projects gain in popularity, their user-bases expand, or their data sizes increase, network requirements can easily become more demanding, forcing projects to upgrade both their computer hardware and network capacities to cope with increased demand. In these scenarios, the centralized data architecture currently employed by BOINC (see §2.3.1) and other Desktop Grid systems (see §2.3.2) can be potential bottlenecks when tasks require large input files or the central server has limited bandwidth. With new data management technologies, it will be possible to explore new types of data-intensive application scenarios — ones that currently are overly prohibitive given their large data transfer needs (see Table 2.4). There are many applications that could either expand their current problem scope or migrate to a Desktop Grid environment if the middleware had support for scalable data management.

Peer-to-peer data sharing techniques, like those presented in §2.5, can be used to introduce a new kind of data distribution system for volunteer and Desktop Grid projects - one that can provide a flexible network overlay that distributes load, and could potentially take advantage of plentiful client-side network capabilities. In addition to serving Desktop Grid computing needs, P2P networks could be used as a migration mechanism to transition Service Grid data to Desktop Grid environments. Figure 3.1 shows an idealistic view of how such a P2P network would be viewed by each target application domain – simply as a “black box” where data can be stored and retrieved.

This functionality could be implemented in a variety of forms, ranging from BitTorrent-style networks (see §2.5.4.2) in which all participants share relatively equally [87], to more constrained and customizable unstructured P2P networks where fine-grained data distribution and discovery strategies could be explored. However, adapting P2P-style data distribution to Desktop Grids is not without its difficulties. Desktop Grid environments have differing needs than general file-sharing P2P communities, primarily because security can become more of a complex issue than solely guaranteeing data validity. Also, scalability needs to

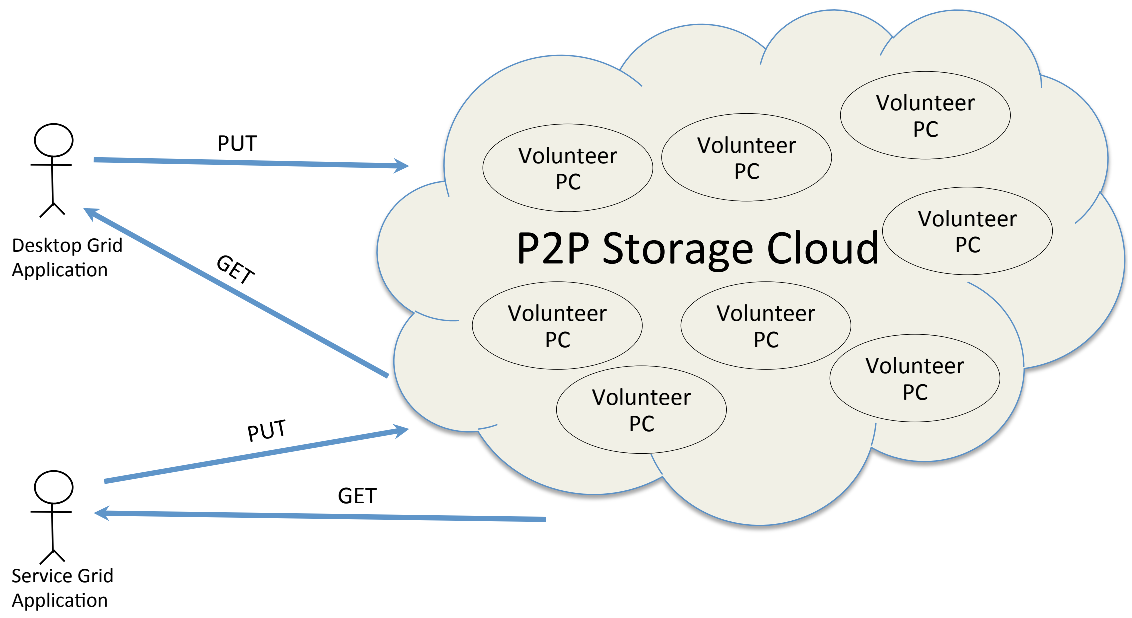


Figure 3.1: Idealistic P2P network for Service and Desktop Grids.

be achieved with the unique characteristic that nodes cannot be forced to act as data providers to be consumers (as required in many existing P2P networks). Additionally, in Desktop Grids, it can be a requisite that only certain amounts of data are shared with an individual peer, further limiting the ability to adopt existing "fair" distribution strategies that would self-build and balance in a traditional P2P network.

This chapter explores the requirements for volunteer computing and how they shape the decision-making process that leads to a suitable system. Beyond the scope of distribution *within* volunteer computing environments, this chapter also analyzes the issue of file *transition* between Service and Desktop Grids. Although volunteer computing is the primary use-case analyzed in this chapter, it should be noted that many of the principles, requirements, and design concepts are applicable to other distributed application domains such as Cloud computing.

For the EDGeS (see §2.4.2) and EDGI projects (see Appendix B for more information), creating a hybrid computing environment that incorporates both Service and Desktop Grids is fundamental requirement. However, there exist several technological hurdles that arise when trying to transition files between these two infrastructures. Chief among them is how to properly distribute the sometimes large (in Desktop Grid computing terms) input files to the highly distributed worker nodes for processing.

Unlike in a Service Grid environment, Desktop Grids do not have a shared high-speed disk or high speed interconnects. It is necessary for each input file to be individually transferred over a Wide Area Network (WAN) to the compute node that will process the job. For the vast majority of BOINC projects, initial data staging is done by central project administrators who have direct access to all input data and replicate it to a series of mirroring Web servers. However, in the case of EDGeS, this transitioning must be incorporated into standard command-line tools, as the input data cannot be known *a priori*, and must be deployed directly by end-users, since ultimate job submission lies in their hands.

For the purposes of the research presented here, an initial hypothesis has been formed which states that distributing data in a self-organizing peer-to-peer fashion is a suitable and useful mechanism in Desktop Grids and for Service to Desktop Grid migration. To prove this hypothesis a new network paradigm is being proposed. Part of designing a new system is to analyze the user requirements and evaluate current state-of-the-art in the field for potential incorporation and/or adaptation. Based on this analysis, this chapter then explores custom solutions tailored specifically for the volunteer computing environment. To validate the choices made and show their applicability to the problem set and ability to satisfy real-world requirements, simulations are then performed on the chosen network overlays. Lastly, this chapter gives the initial network design of a peer-to-peer data management system for Desktop Grids.

3.3 Application Environment

The BOINC architecture[1] is based on a strict master/worker model, with a central server responsible for dividing applications in thousands of small independent tasks and then distributing the tasks to participants, or worker nodes, as they request work units. To simplify network communication and bypass any NAT problems that might arise with bidirectional communication, the centralized server never initiates communication with worker nodes; rather, all communication is instantiated from the worker when more work is needed or results are ready for submission. In the current implementation of BOINC, data distribution and scaling is achieved though the use of multiple centralized and mirrored HTTP servers

[1] XtremWeb data requirements are very similar to BOINC. Therefore, in this chapter, data sharing references and solutions for BOINC can be seen as equality applicable to the XtremWeb architecture.

that share data with the entire network (see §2.3.1 for more information).

The centralized architecture of BOINC creates a single, or in the case of mirrored servers, a small number of failure points and potential bottlenecks and fails to take advantage of plentiful client-side network bandwidth and storage capabilities. If broader resources could be successfully used to distribute data sets, not only would it allow for larger data files to be distributed, but it would also minimize the needed central networking capabilities. This would substantially lower the operating costs of many BOINC projects, while allowing for a broader range of applications to leverage volunteer computing. To decentralize the current data model of BOINC, as well as provide load balancing for XtremWeb, a peer-to-peer data distribution approach is proposed here.

When considering the practical application of P2P technologies to the "production" BOINC environment, several concerns must be adequately addressed if the solution is to be successful. For the purposes of this research, the following four are given priority. Section 3.4 explores them in more detail.

Scalability and Network Topology — Ability must exist not only to adapt on the Wide Area Network (WAN), but also to detect and exploit Local Area Network (LAN) topologies, using relative proximity and response times. The ability to hone network topologies and requests based upon proximity is a useful way to further limit the amount of necessary bandwidth needed to serve project files, as well as limit broader (i.e., Internet) network congestion. The trade-off is generally that the looser the system becomes in its ability to adopt and utilize network proximity (such as providing caching nodes on LANs), the more complex the system becomes and prone to abuse and potential misconfiguration.

Data Integrity and Security — A mechanism for identifying hosts that supply bad data, and subsequently banning them from the network or having ways to avoid using them (i.e., through de-prioritization), must be included to ensure network stability. The issue of which nodes are able to propagate data on the network, and therefore have the ability to inflict damage, should depend upon the individual policies of each hosting project, to give flexibility to a number of scenarios and project requirements. In the most restrictive case, only trusted and verified participants would be certified to propagate data. In more lax security configurations, which exploit a larger pool of resources, security would have to be more flexible. Regardless of the decision, data signing can help to prevent analysis of corrupted data, leaving the primary concern being how to limit network flooding by “bad” providers.

User Security and Client-Side Configuration — A peer-to-peer infrastructure should protect the users and ideally have a way to automatically configure routers or bypass NAT issues. Depending on an individual project’s configuration, firewall and router issues could be problematic, with a general trade-off between “punching holes” in clients’ firewalls to exploit their bandwidth and the security concerns of doing so, as well as the extra software development and configuration this demands. In volunteer computing projects it is especially important to provide a high level of security to participants. If NATs are bypassed, it must to be done in a secure and transparent (to the end-user) manner.

Legacy Software Integration — Any new technology for Desktop Grids must integrate with the current client and server software. For the purposes of the research presented here, this equates to integration with BOINC and XtremWeb. It is important for any software that wishes to provide an added value to the larger BOINC and XtremWeb communities to have little or no impact on current operating procedures. Requiring external libraries or other similar dependencies could prove to be problematic and greatly limit uptake. The BOINC client is currently written in C++; any successful add-on would most likely have to adapt to this requirement. XtremWeb is written in Java, and would likewise require compatible libraries to send and receive data.

It is useful to note that the above considerations are not exhaustive of all research topics that could be explored when designing a new data management system. Rather, they represent a subset of topics that have been identified for this work, each of which having a direct impact upon the target end-user community. The selection process for these traits was to view the problem domain from the standpoint of a volunteer computing application developer, and seek the practical steps that are needed to employ a successful system. Other important issues such as fault-tolerance (also integral to any data management solution, and a thesis topic in and of itself) and the management of small file sizes outside the scope of volunteer computing are intentionally omitted, allowing the work presented here to be scoped.

3.4 Design Considerations

This section discusses in further detail the four target issues raised in §3.3 that have been identified as useful when considering a new data distribution paradigm for volunteer computing. Note that the list presented here is not exhaustive and does not seek to provide an all-encompassing taxonomy of every potential metric for designing data distribution networks that are optimal for every Desktop Grid application. For example, the size of the network can vary dramatically between the extremely popular BOINC projects and their less successful counterparts, which could result in a high-performance solution for one being less than perfect in another situation. For each project within BOINC, these factors will vary slightly and these differences can make designing an optimal network for all of the BOINC community a challenging task. Yet as shown in [88], the application of a P2P network layer would allow many additional and unused network and storage resources to be leveraged by BOINC projects without sacrificing necessary processing power. Even a generic solution should still have an overall positive impact.

Therefore, rather than identifying and attempting to solve every issue for every application, this section isolates a small number of the broad and general factors that become important for the majority of applications. These factors are significant when designing and deploying a data-serving network in the domain and across the large scales such as those seen in the BOINC community.

For the research presented here, the following discussion, then, is seen as the base litmus test for the selection and development of the underlying network technology and topology. The metrics identified here show the principal components that are crucial in the development of a new system that is suitable for a wide range of Desktop Grid applications. Concurrent to this exploration, the need for translation paths from Service to Desktop Grids is accounted for, in order to provide greater impact and usability by the EDGeS and EDGI projects.

It should be noted that for optimal matching and highest performance, one should go beyond these principal metrics, and further fine-tune the architecture presented in Chapter 4 to their individual application needs and environment. However, any such further optimization for specific projects or alternative metrics is considered as outside of the scope of the research presented here.

3.4.1 Scalability and Network Topology

There are a number of scalability considerations for data distribution in a volunteer computing environment. This is especially true given the large number of nodes that can join the network, compounded with the size and sensitivity of data. Too little flexibility of the system to scale can lead to an overload on data serving nodes, while too much flexibility can entail looser security criteria and policies with regard to which network peers are allowed to propagate and cache data. Most current projects in BOINC have limited data needs for an individual work unit (e.g., a few megabytes); however, they still require large network bandwidths and distributed caching due to the sheer size of their volunteer networks. For example, the SETI@Home project, which has relatively small 340 KB input files, requires ≈50 Mbp/s of consistent upload to serve its ≈233,000 active users (see §2.3.3.1).

A viable alternative for data distribution is to employ the use of peer-to-peer techniques. P2P systems can support the on-demand auto-election (and de-election) of network participants to be data providers, which can enable network scalability as the number of requests expand and contract. The application of using P2P in volunteer computing as a means to offload the central network needs has been explored in [85]. There are many ways this could be implemented, ranging from a BitTorrent-style network [65], where data is centrally tracked (i.e., potential bottleneck) and all participants share relatively equal loads, to Kazaa-like super-peer networks [86] where select nodes are assigned greater responsibility in the network and there is no central authority.

Beyond the ability to scale network throughput and capacity, the above example showing SETI@Home requirements raises the issue of appropriate network topologies. Recall the discussion in Chapter 2 that presented a peer-to-peer file distribution taxonomy (see §2.5.2). With hundreds of thousands of active users, flat networks become unrealistic, as proven by the example of early Gnutella versions. Tightly coupled and LAN-oriented systems are likewise irrelevant, as they were not designed to operate over wide area networks and would perform poorly. Therefore, when seeking a P2P solution, it becomes necessary to look at the remaining approaches of *Centralized/Decentralized* and *Distributed Hash Tables* (DHTs).

Since within volunteer computing environments it is unlikely that all network participants will be sharing data, a direct use of distributed hash tables, which would require all nodes to be part of the content network, is not possible. In addition, for most Desktop Grid applications (as well as for this research) the problematic issue is data distribution to worker nodes, generally not long-term data storage. Therefore, creating a network that broadly uses DHTs between all peers to map discrete data to known locations is a less interesting task than managing initial data distribution.

In any realistic scenario for volunteer computing, only a subset of the network would be able and required to store and share files; otherwise adoption would be severely limited. Therefore, the initial topology must be a hybrid or centralized/decentralized network, with some peers having more responsibility (e.g., acting as data repositories) than others. DHTs could still prove useful as a way to locate and distribute data among the data sharing overlay, as shown in [89]; however, their adoption then becomes an implementation and optimization issue rather than a core network design requirement.

Using the above logic to implement the data-sharing aspects of BOINC would require a new overlay network to be created that contains only those nodes that have been chosen to act as data providers. This is very similar in concept to the current static mirrored system that most BOINC projects use (i.e., a subset of available resources performs additional work), however, with the fundamental difference that the selection of mirrors becomes dynamic and decentralized, while allowing for greater caching flexibility beyond full-mirroring of an entire dataset. In addition, within the data sharing overlay, nodes could propagate data among themselves, rather than relying on direct synchronization with the raw data source. By partitioning files and allowing individual segments to be concurrently distributed across a larger network, members of a data caching overlay can be both consumers and providers of the same file, greatly increasing throughput through a concept called “file swarming” (see Figure 3.2). Additionally, this process can be further honed to use more advanced distribution algorithms such as FastReplica [90], which allow for the exploitation of varying Internet paths.

In addition to being able to adequately serve the data needs of the network and efficiently distribute files, any P2P system for Desktop Grids needs to be able to support a network topology that allows projects to restrict the data sharing layer based upon their own individual security criteria and needs. This is necessary because many projects do not want arbitrary nodes to be able to cache large amounts of their data, in addition to the general security implications of allowing anyone to act as a data provider, where he/she could flood the network

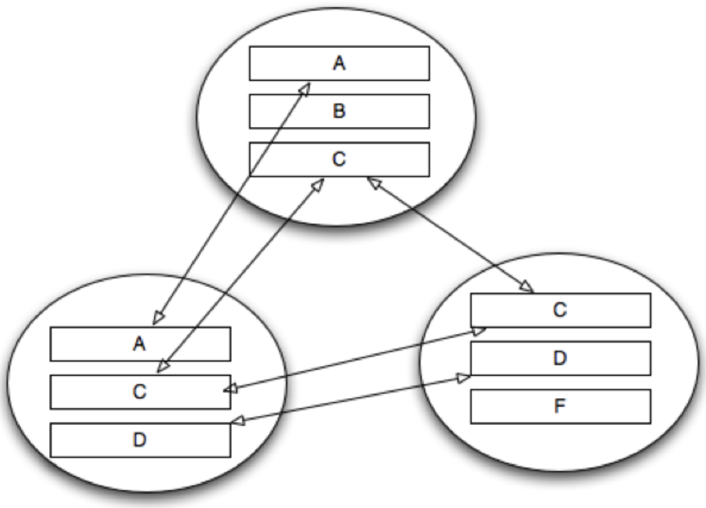


Figure 3.2: Data Center File Distribution — shows how Data Centers can share files (and verified partial files) among one-another, speeding distribution.

with bad requests and cause unnecessary congestion. Such restriction can be implemented through the application of a “trusted” overlay that provides lookup services to find data and controls which peers in the network are able to register as data providers — thereby controlling data distribution policies through a restriction of data advertisements.

3.4.2 Data Integrity and Security

Data security, especially when dealing with scientific and commercial data, can be a complex matter. At the most basic level there is the issue of file integrity, or ensuring that files have not become corrupted or manipulated in transit. Verifying file integrity is a fairly well-understood subject area and there are suitable solutions to this issue, such as validation of file authenticity through signing and comparing hashes from the downloaded file with those provided by the project’s (trusted) central authority. To limit the effect of malicious or badly configured hosts, larger files can be split into smaller discreet “chunks” which are then individually hashed and verified, as seen in Figure 3.3. Such segmentation allows clients to detect bad data early on and blacklist offending hosts. After all chunks are downloaded and verified, clients reassemble the resulting file and verify it in its entirety before proceeding with job processing.

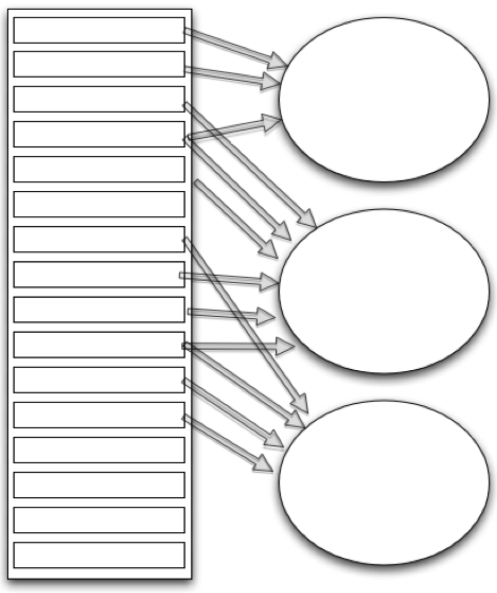


Figure 3.3: File Segmenting and Swarming — shows how files can be split into distinct segments. This allows for intermittent checking of data integrity as well as partial and quicker replication across the Data Center overlay due to double hashing.

When building a P2P network for volunteer computing, there are a number of security requirements beyond the traditional notion of simply ensuring file integrity. Due to the volatile and insecure nature of these networks (e.g., BOINC and XtremWeb), a product of their open participation policies, there can be reason to enforce limitations on which entities are allowed to distribute and cache data. Especially when opening the data distribution channels to wider (possibly public) participation, security can become a concern. Since file integrity can be sufficiently guaranteed due to the centralized nature of data origins, the more interesting research questions in the realm of data security become those of how to verify and authenticate the different network agents that will be propagating information and/or caching input data. This analysis can be broken down into the two broad subject areas of *authentication* and *authorization*.

Authentication is the verification process by which an entity identifies itself to others and gives evidence of its validity. Public Key Infrastructure (PKI) [91, 92] is a proven tool that can be fairly effective for performing peer identity authentication. In the simplest case, this can be done by having a central authority (e.g., the BOINC manager) sign and issue either full or proxy certificates to those it deems trustworthy enough to distribute data on its behalf.

When another peer on the network contacts this “trusted” entity, it can use the public key from the centralized BOINC manager to verify the authenticity of the trusted peer. This process can likewise be performed in reverse, provided clients are also issued certificates, as a means for the data distributers to validate the identity of the clients and verify that they have the proper credentials to retrieve data. The process of using certificates for mutual authentication can provide individual peers with certainty that the host from which they are retrieving data has been delegated the proper authority and *visa versa*.

Beyond file integrity verification, more advanced and customizable use-cases that provide for interaction between multiple virtual organizations and hierarchal delegation (e.g., certificate-chaining and cross-certification agreements) can be derived from this simple arrangement [92, 93]. Although these offer the ability to compose advanced workflows and promote delegation of authority, they are beyond the scope of this research and not deemed necessary to implement a suitable data distribution network for straightforward master/worker bag-of-task applications on Desktop Grids.

Authorization is a much more interesting question than authentication. This is primarily because there are standardized techniques for authentication that can be widely applied to many different applications with little or no modification. Authorization, conversely, is application-specific, differing with each individual application’s unique needs and authority structures. Although there are tools to help define authorization policies and enforce them [94], the policies themselves will be different with each application.

At the most basic level of authorizing select peers to cache and distribute data as they see fit, authorization can be straightforward and unproblematic. For example, it is possible to issue certificates (e.g., X.509) to the data cachers (as mentioned in the authentication section) that would then allow them to be validated as data distributers and authorize for tasks by mapping their Distinguished Names (DNs) to roles. So long as each data caching host has equal access rights relative to all other data cachers, solutions like this can prove useful; however, they can become limiting if more advanced solutions are needed that partition standard CRUD (Create, Read, Update, Delete) operations.

To differentiate between who can create, read, update, or delete files, more dynamic and customizable architectures are needed. It is when more dynamic and customizable queries are needed, such as fine-tuned control over specific data and its caching policies, that the management of an authorization infrastructure and Access Control Lists (ACLs) can quickly become complex. Thus, depending on application requirements, the issue can require a higher level of diligence, and any scheme that goes beyond a simple yes/no query must be customized specifically to the target environment and its individual policies.

For example, in the BOINC environment, possible ways in which a reasonable level of authorization could be implemented would be to tie certificate issuance into the centralized scoring system that keeps track of users and groups that are contributing cycles to the project. In this scenario, when a new potential data server enters the network, it would contact the group of data servers in the network and offer to join them. After certificate exchange to authenticate identity, a lookup could be done against the scoring system to see if the newcomer meets the requirements to join the data center network. If this was successful, the newcomer could then be added to a centralized or decentralized list of “authorized data providers.”

If this were implemented in the absence of a centralized authority, for example, in some XtremWeb scenarios where no central credit system exists, one issue that would arise would be how to continuously validate the data center layer and remove data centers that have turned rogue or been compromised since joining the network and passing the initial validation step. One potential solution would be to issue only proxy certificates that expire after a given amount of time, thereby requiring periodic re-authorization. Then when clients connect to a stale data center, they could verify the lifetime and expiration date of the certificate.

3.4.3 User Security and Client-Side Configuration

User security refers to the ability to protect the “volunteers” in the network from any harm that could possibility be inflicted on their machines by another network participant. This is a very important issue when one is within the realm of volunteer computing, as the resources typically are donated home computers that likely contain volunteers’ important personal information and documents. In the case of a security breech in which these volunteer resources were compromised by some malicious entity, the potential fallout could be enormous. Fear of a harsh backlash has been one of the limiting factors to the incorporation of standard P2P technologies into the BOINC middleware. Even in the event where no actual security breech takes place, *requiring* peers to share data with one another via P2P protocols, such as with BitTorrent which enforces sharing, could have the downside of alienating potential volunteers. This can be due to a number of factors, ranging from a volunteer's unwillingness to donate network resources (perhaps due to bandwidth requirements from other computers on the same network or a metered data connection) to misconceived public perception that associates peer-to-peer technological implementations with some of the more controversial uses of the technology (e.g., illegal file sharing).

In addition to the sociological issues, there are also technical hurdles to be overcome when incorporating client (personal) computers into a data sharing network as data providers. The asymmetric up/download speeds of most consumer Internet connections generally relegate uploading bandwidth to a small fraction of the download speed. How to route data through Network Address Translation (NAT) systems is another key problem since most home networks lie behind a router (a result of the limited IPv4 address space) and are not directly contactable by other Internet entities. Even in the case where these obstacles can be overcome, for example through using uPnP or proxy servers, personal firewalls often have to be configured by end-users to allow external connections.

As data consumers, security implications are less pronounced than when acting as data providers. To retrieve data from the network, clients do not need to expose their machines to outside connections. Rather, they initiate *pull* requests to data providers, thus requiring firewalls to be minimally configured for outbound access only (and maintenance of established connections), as is the norm. The use of file swarming, as depicted in Figure 3.4, allows clients to exploit multiple network paths, as well as to leverage more than one data endpoint to maximize download speeds. Such a system does expose clients in the sense that the data providers know their identities and could subsequently attack them or provide bad data. However, as shown in §3.4.2, bad data can be marginalized and in this scenario, exposure to foreign hosts is no different or more risky than any commonplace Internet browsing experience where IP addresses are exchanged.

When considering user security, it is necessary to be cognizant of the notion that the BOINC and XtremWeb communities rely upon volunteers to function. Any new data distribution scheme that is implemented must allow users to opt out if they do not wish to share their bandwidth or storage capacity. Even in

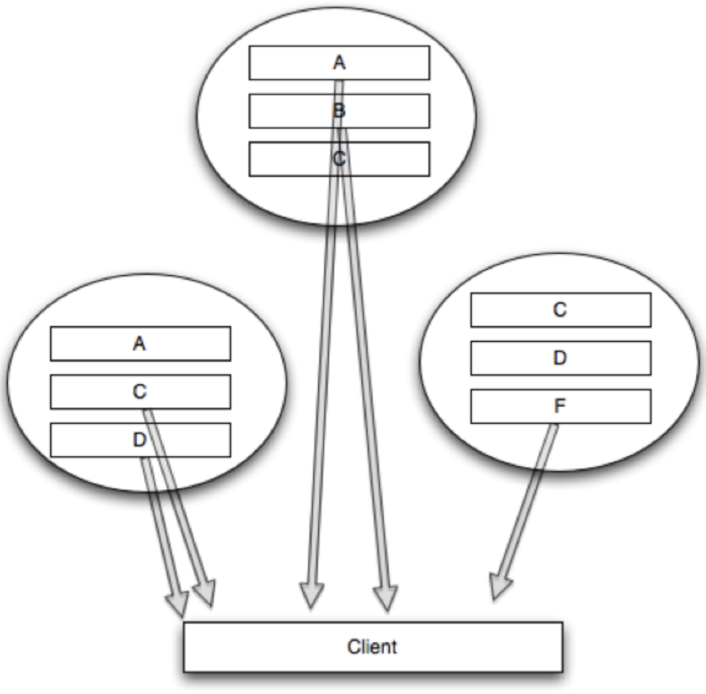


Figure 3.4: File Swarming to Client Machines — shows clients downloading from multiple Data Centers, which exploits network paths, distributes load, and potentially addresses asymmetric speed distribution.

the instance where users have opted to share data, a high-level of consideration has to be given to ensure that their computers are adequately protected from attacks. In BOINC environments this is currently solved by having a centralized, and presumably non-hostile, authority that distributes both executables and data. In XtremWeb, it is done through mutual trust of the community: that malicious users will not exploit network resources. Even in these scenarios, there is still a chance that servers could be compromised, or that executables have inherent security flaws. Such risks are generally minimal and if and when they occur, would likely be the consequence of inaction on the part of application stakeholders (i.e., projects) to pursue a more rigid security policy.

3.4.4 Legacy Software Integration

Beyond the issues discussed in the previous sections, there is an additional very important factor that needs to be considered for application uptake and longevity: the ability to integrate with existing Desktop Grid software. As shown in §2.3,

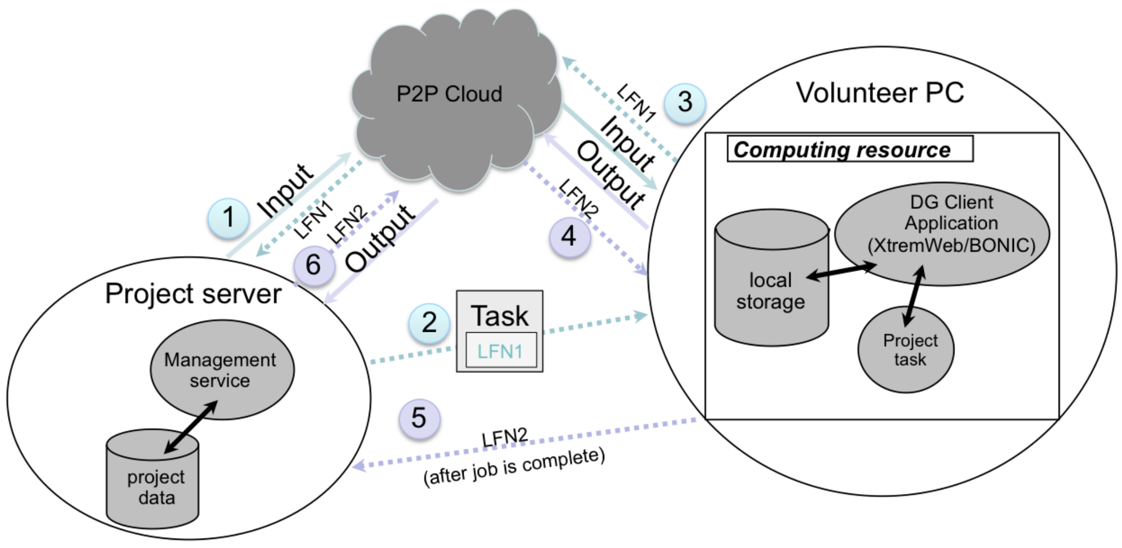


Figure 3.5: High-Level P2P View — shows how a "P2P Cloud" could manage file distribution for project servers.

there are several dozen Desktop Grid projects actively operating that use either the BOINC or XtremWeb middleware. Each of these projects shares a common thread with one another: each has a highly parallel problem that can be split into thousands of independent tasks and asynchronously processed. It is these properties that allow the projects to exploit a Desktop Grid environment and utilize the numerous volunteer computing resources that are made available in the process. What isn't apparent, however, are the vastly different levels of data intensity within the projects. This can manifest itself in the form of varying data input and output file sizes, changing replication facts and different throughput requirements.

Ideally, a data distribution network for Desktop Grids would be one where both input/output data could simply be pushed/pulled from a data management "black-box" which is self-organized and balanced based upon network demands and availability.

A conceptual workflow, as shown in Figure 3.5, illustrates how data could be sent and retrieved from such an idealistic P2P network, described as follows:

1. Input data is published to the network and a Logical File Name (LFN) is returned. The data-sharing network would then self-replicate the data for distribution.
2. The logical file reference, *LFN1*, is injected into workunits as the location of input data. The workunits are sent to the volunteer PCs for processing.

3. Client machines use the *LFN1* reference to retrieve the concrete file from the network. Once download is completed, the workunit can be started.
4. Output data is likewise published to the network, with the client receiving a new file reference, *LFN2*, corresponding to its published file.
5. *LFN2* is returned to the project's servers as part of the completed job's metadata.
6. The project server can then retrieve the concrete file referenced by *LFN2* at any time, or leave it on the network for storage.

Beyond data intensity, projects also differ in which Desktop Grid middleware they are using. For the purposes of this search, that equates to BOINC or XtremWeb. The choice of middleware determines how data is hosted and distributed throughout the project, as well as the native programming language that any adapters or enhancements to the software in which it would need to be implemented. In the case of BOINC, client software is C/C++ based and any enhancements to BOINC should likewise be in those languages, as it cannot be expected that a Java Runtime Environment (JRE), or other library, is installed on clients' machines. Conversely, Xtremweb is written in Java, and therefore any enhancements to XtremWeb would also need to be implemented in a way that eases integration.

One fundamental difference between deploying a new solution for XtremWeb and BOINC is the size of the existing network. XtremWeb is a more dynamic piece of software than BOINC, with a smaller group of active clients that could likely be updated to a new version of the software if additional features warranted the effort. It is likely, therefore, that enhancements and the addition of new libraries could be built-in to a new release cycle. BOINC, on the other hand, has hundreds of thousands of active volunteers, many of which do not often update their software. Backwards capability is key for BOINC, or finding ways to retrofit a new solution into old clients, without requiring any client-side modification.

3.5 Analysis of P2P Network Architectures

Applying a P2P data distribution approach could be achieved in a variety of forms. The most straightforward method would be to adapt a current P2P technology to

Desktop Grids. This approach was explored (in collaboration with colleagues from Coimbra and INRIA) in [95], where a modified BitTorrent[2] was used to distribute BOINC files, and in [89], where a DHT-based Content Delivery Network (CDN) variant was analyzed.

In this section, the results of the analysis of those two different approaches are given, as well as the proposal for a new network paradigm that combines the strengths of these technologies, while limiting network function and complexity to the necessary components to implement a useful and novel solution for Desktop Grids.

3.5.1 BitTorrent

For the BitTorrent experiments, input files were published to a BitTorrent network, and the location of the *.torrent* metadata file is injected into workunits, in a style similar to that proposed in the first three steps of Figure 3.5. The centralized BitTorrent tracker, as seen in Figure 3.6, keeps a list of which entities on the network have a given file. When a client wishes to retrieve input, it parses its BitTorrent metadata file (*Step 1:* File.torrent), contacts the tracker (*Step 2:* Tracker), and receives a list of download endpoints. File transfer then begins, with the client receiving small segments of the input data from several servers (*Step 3:* File.data), and (most likely also) registering itself as a data provider to concurrently share completed segments of the file with others on the network.

The analysis proved very positive towards using BitTorrent, or another similar P2P software middleware, as a distribution mechanism. The BitTorrent tests showed a significant speedup in overall download time coupled with a significant reduction in load on central data severs, with very little computational overhead. BitTorrent was shown to be able to fairly effectively solve the data needs of BOINC as they relate strictly to distribution. However, it has limited security beyond ensuring file integrity and has no notion of grouping or peer hierarchy. An out-of-the box BitTorrent solution also constrains the system to one in which clients are active data providers, a requirement that makes it an unsatisfactory match for the requirements identified in §3.4. It is also difficult to implement due to private networks and client policies. Fortunately, the technologies and techniques used in

[2]BitTorrent was chosen because it has proven to be both scalable and efficient and could be especially beneficial to projects that have large input files that can be shared between numerous independent workers. See §2.5.4.2 for more details.

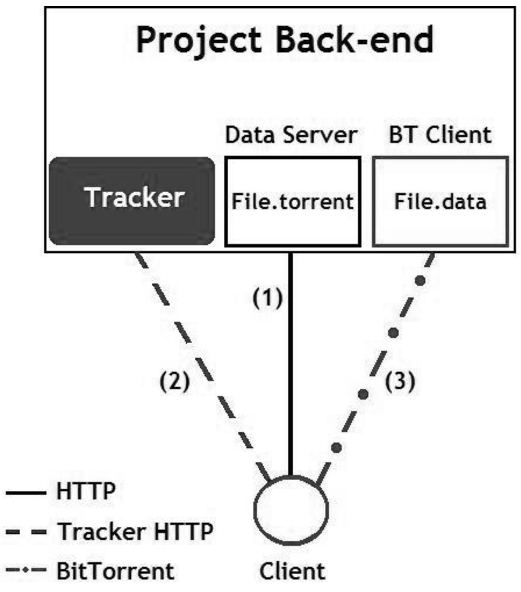


Figure 3.6: BitTorrent and BOINC — shows how BitTorrent could be used with BOINC to distribute files.

BitTorrent, such as file swarming, chunking, and double-hash verification, are extremely useful and relevant and the positive analysis of BitTorrent proves as a fundamental building block for identifying useful features for further data distribution methods.

3.5.2 Super-Peer Topologies

Similar to the analysis of the BitTorrent example, a solution for Desktop Grid data distribution was sought in "super-peer" networks (see §2.5.6). These networks allow active participation by numerous hosts, but do not require role equality, and necessarily enforce a strict BitTorrent-like "tit-for-tat" policy to ensure network parity and balance load. The use of super-peer networks, as shown in Figure 3.7, and their applicability to Desktop Grids was analyzed (in collaboration with ICAR-CNR) in [96–99]. In the aforementioned research, a traditional super-peer network topology is explored, in which many peers act as relays for messages and data traffic, in a method very similar to that employed by later versions of Gnutella (see §2.5.6.1). In such a system, responsibility is often dynamic and entities can perform more than one function, with promotion and demotion into different role categories, depending on network need.

Although initially attractive as a means to manage network load and construct

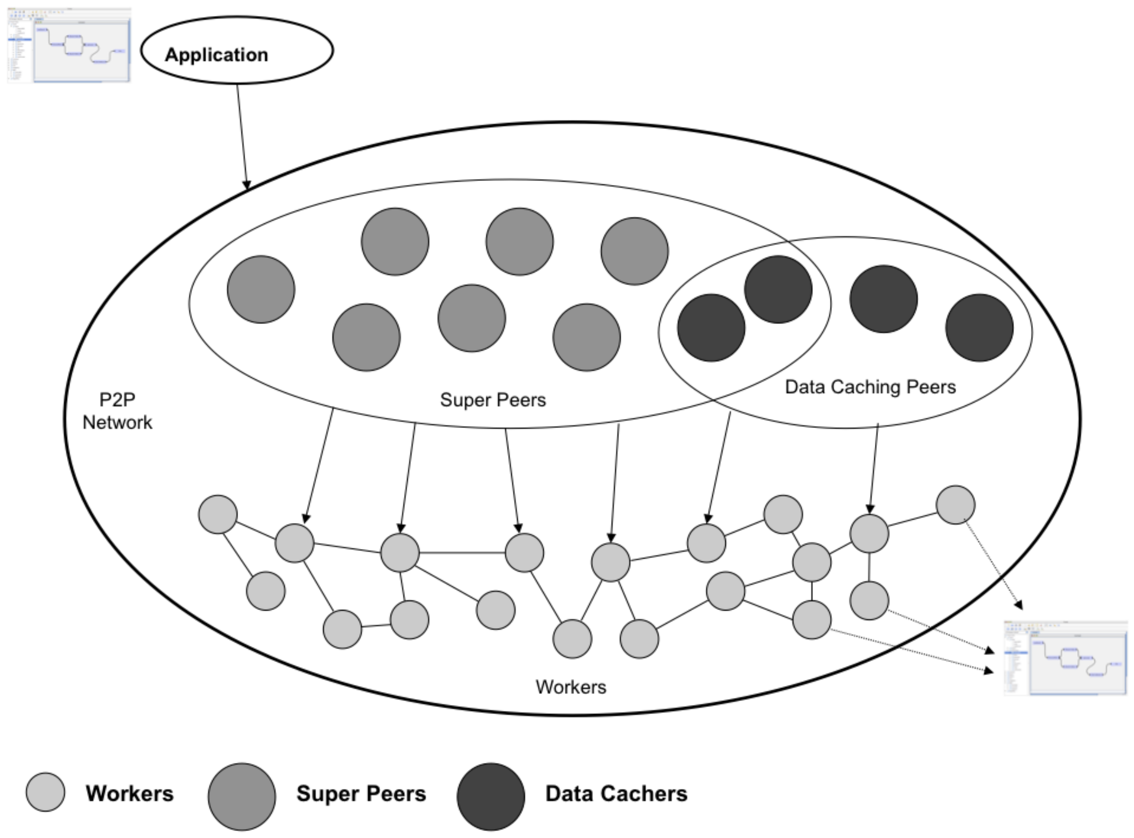


Figure 3.7: Super-Peer Network — shows how some roles overlap and entities can have different scopes, network connections, and responsibilities.

a P2P network, this approach suffers from the complexity of message overhead, lack of centralized control, and, if one wants to adopt a pre-existing super-peer technology, there is the cost of “buy-in” to their network building concepts and implementations. After the success of simulating super-peer networks, Peer-to-Peer Simplified (P2PS) [100], and JXTA [101] were explored as potential middleware to construct a new data sharing network. Although both P2PS and JXTA provide generic tools for building super-peer networks, they proved to be limiting either in their ability to scale or to form role-based groups where the developer can explicitly form the topology and control message relaying without major modifications. Specifically, P2PS and JXTA were abandoned because of two main reasons. First, neither allow the fine-grained access controls needed for the data layer. Second, there were no caching policies in either system for data rather than metadata (adverts or queries). Therefore, the data layer would essentially have to be built from scratch, meaning that the benefits of either system are reduced to providing their respective P2P abstractions. It was therefore decided that the benefits of using these systems were far outweighed by the drawbacks of the additional dependencies they placed upon the end-user, and their increased complexity.

Many of the traits shown by JXTA and P2PS were not seen as desirable to Desktop Grid applications wishing to leverage P2P file distribution. Although they provided a base-middleware, they added additional complexity and did not necessarily reduce development or design time. In addition, it was seen as undesirable to relinquish control over network and data propagation to the constraints imposed by existing systems. These, and other loosely coupled super-peer networks, required overly complex security mechanisms to construct their overlays (if security is desired), through the use of distributed certificate management and authentication protocols [93]. Lastly, the given middleware for super-peer networks was notably in the form of low-level frameworks that still require significant and nontrivial programming modification to construct a functioning system. Therefore, although super-peer networks were deemed capable of performing the tasks needed, they suffer from too much complexity and required substantial modification.

3.6 Conclusions

Chapter 2 gave an overview of many of the popular and proven P2P systems that use different methods to enable data distribution. There exist also commercial solutions like Amazon's S3 [19] or Google's file system (GFS) [79], that could be fairly effectively applied to provide for the data needs of BOINC or XtremWeb, at least as they relate strictly to distribution. However, in the case of commercial products, there is a direct monetary cost involved, and for P2P systems like BitTorrent, the facility to secure or limit who is able to receive, cache, or propagate different pieces of information is generally limited or nonexistent. These limitations make it difficult to directly adopt one of the above as a product for data distribution for scientific applications such as BOINC, however, they do provide a good basis for building a new technology that conforms to the project requirements.

This Chapter has focused on defining the requirements and scoping the application domain for Desktop Grid data solutions. Current technologies were investigated to find a suitable "out of the box" match for data distribution. Through the process of evaluation, it become apparent that in volunteer computing environments, policies and safeguards for scientific data and users' computers become more critical concerns for limiting update rather than any technical feasibility. Scalability becomes the balancing act of applying a highly scalable system (e.g., Kazaa) with a reliable and well-defined central authority that can regulate the system. A tailor-made solution that could take into account the requirements of scientific communities, as opposed to a generic overarching P2P architecture, would have the advantage of facilitating different network topologies and data distribution algorithms, whilst retaining the safety of each participant's computer. Further, each scientific application has different network and data needs, and customized solutions would allow for tailoring the network towards individual requirements, albeit with the disadvantage of increased development effort, complexity, and code maintenance.

Although even in this scenario, there are still chances that the servers could be compromised, or that the executables distributed have inherent security flaws, this is generally a very minimal risk and would be a consequence of actions of the application stakeholders, not third-party unknown distributers. It is these considerations and requirements that make applying P2P protocols such as BitTorrent, which enforce tit-for-tat sharing, problematic, as the specification is explicit in its requirement that all peers must share in order to receive, exposing network participants to harm or misuse. This is a very important issue when one is within the realm of volunteer computing, as the resources are typically donated home computers that likely contain volunteers' important personal information and documents. In the case of a security breech in which these volunteer resources were compromised by some malicious entity, the potentially fallout could be enormous.

Fear of a harsh backlash has been one of the limiting factors to the incorporation of standard P2P technologies into the BOINC middleware. Even in the event where no actual security breach takes place, *requiring* peers to share data with one another via P2P protocols could have the down-side of alienating potential volunteers. This could result from any number of factors, ranging from a volunteer's unwillingness to donate network resources (perhaps due to bandwidth requirements from other computers on the same network or a metered data connection) to misconceived public perception that associates all peer-to-peer technological implementations with the more controversial uses of the technology.

A new solution that combines the flexibility and dynamic nature of super-peer networks, with the simplicity of BitTorrent, seemed to be a promising match for Desktop Grids. As we have seen from the earlier requirements analysis, in volunteer computing communities, security can be a much larger issue than simply

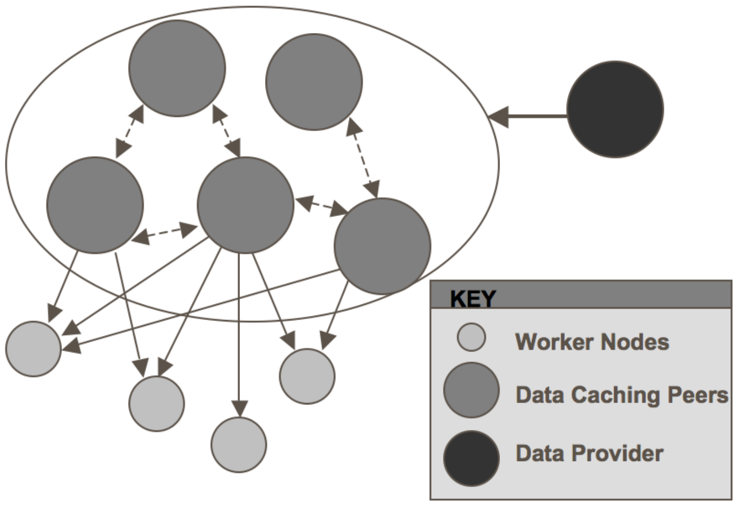


Figure 3.8: Proposed Network Topology — gives a snapshot of a simplified network topology with a limited number of roles. Data Centers share data amongst themselves and respond to Worker-node client requests.

guaranteeing data validity, and file distribution strategies can be more complex than providing full replica mirroring. A successful network infrastructure needs to be a more flexible solution than BitTorrent, and offer a more straightforward and easily governed network topology than a traditional super-peer topology. The ultimate goals should be to minimize network entities to core functions, limit message traffic and upkeep due to churn, provide a central authority that can easily be governed and managed, and enable some form of network segmentation into *providers* and *clients* to facilitate volunteer computing needs — and all while still achieve effective distributed data propagation.

Figure 3.8 gives a visual representation of how a more simplified network could look. Here, a single overlay provides data distribution, in contrast to the more complex super-peer network shown in Figure 3.7, where some nodes act as message overlays and may or may not serve data given a particular request chain. The design of a new "Architecture for Data Intensive Cycle Sharing" (ADICS) is introduced in the next chapter. ADICS is constructed with adherence to the design principles introduced here. Its new design proposes a "Data Center" network that eliminates the message-relaying super-peer overlay (like those found in Gnutela) and favors a Napster-inspired topology, where a central authority controls metadata distribution and file registration. As will be seen in Chapter 4, this new topology matches well to the requirements of volunteer computing environments, while still leveraging the power of P2P data brokering.

CHAPTER 4

A Peer-to-Peer Architecture for Data-Intensive Cycle Sharing

The Data Center network organization introduced in this Chapter is a Peer-to-Peer **A**rchitecture for **D**ata-**I**ntensive **C**ycle **S**haring (ADICS) [102, 103]. ADICS offers itself as a data distribution paradigm that can be used to balance network loads and offset storage requirements for highly parallel scientific applications. It is specifically targeted at meeting the challenges involved in volunteer computing environments. The core infrastructure of ADICS is focused as much on environmental considerations as it is on idealized technology. As a software development package and research topic, ADICS is proposed with close consideration of the issues raised in Section 3.4, while at the same time maintaining a desire to construct a core infrastructure that is fairly application agnostic. Therefore it can be used with other applications seeking distributed data management.

Building upon the analysis presented in the previous chapter, here the details of the ADICS design given here, show how and why different infrastructure decisions were reached. It is here that the correlation between design, background, and requirements coalesce to form a new system that not only addresses data transfer within Desktop Grids, but also transition between Service and Desktop Grids. The migration of data between these two different environments is particularly relevant, as it showcases the flexibility of the system and also defines the context within which much of ADICS was researched and developed.

This chapter is organized as follows. First, an architectural model is given that shows the network topology, entities, message types, and the key infrastructure elements in ADICS. Next, the data sharing protocol is given in more detail. The design is then rated against the requirements specified in Chapter 3, paying particular emphasis to the four design issues identified there. Simulation is used next to verify the design and help to fine-tune the protocol definition. Lastly, a first iteration software prototype (the basis for the next chapter's final implementation) is discussed and concluding remarks are given.

4.1 Architectural Model

To achieve a new network design for Desktop Grids that allows for distributed data management, while preserving the simplicity of centralized management, a Napster-style architectural model is proposed. Key to the design presented here is the concept of "Data Centers" (DCs), which serve as the core component in the distribution network overlay. Data Centers act as partial replica caches (see Figure 3.8), with membership to the Data Center overlay (optionally) controlled by a project's central authority. To support file distribution and replica management, the Data Centers exchange files with one another and each is able to serve as a mirror of a segment of the project's input data.

Unlike a traditional super-peer network (see Figure 3.7 in Chapter 3), the Data Center overlay membership is controlled and does not include additional message relaying peers. A simplified view of the relationship between Data Centers and other network entities is given in Figure 3.2. Limiting network entities and constructing the network as a single overlay with shared responsibility is a fundamental design decision that helps to reduce complexity and increase manageability and oversight of the network. Individual members of the Data Center overlay can enforce local policies and preferences regarding how much and how often they provide replica services. At the same time they provide a distributed cache that doesn't require complete mirroring, which is inefficient and greatly increases participation requirements.

Lastly, a Data Center overlay can be augmented with a managed security infrastructure that enables a secure network to regulate the distribution of data based upon project preferences. Therefore, "secure" data centers are a way of implementing a super-peer topology for data sharing that would restrict the set of peers that are allowed to propagate data. In this scenario, policies can be set by each Desktop Grid project as to which participants, if any, are allowed to host and redistribute data. Here a centralized entity can control the distribution and verification of security credentials and roles. Beyond simply restricting Data Center membership, policies can also be introduced to govern the relative sensitivity of data and retention policies. By maintaining control of network membership, new types of functionality and advanced scenarios can be introduced with minimal invasive network tinkering and upkeep.

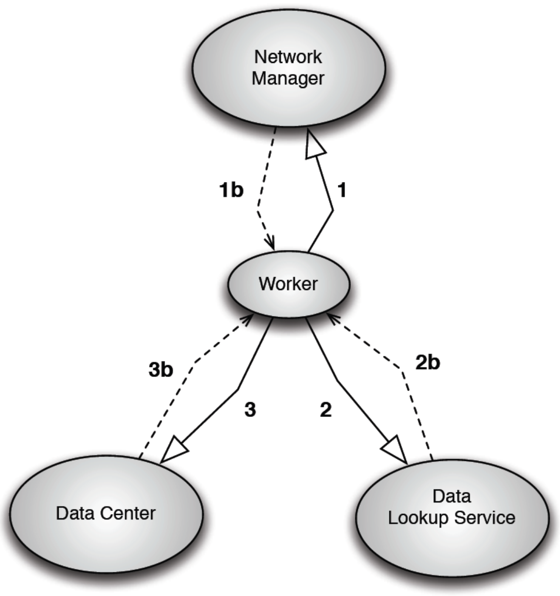


Figure 4.1: ADICS: Client Message Workflow — shows the basic flow of messages from a Worker to various other network entities.

Figure 4.4 shows the basic flow of messages for a client wishing to retrieve data from the network in an ADICS architecture. First, the Network Manager (e.g., BOINC or XtremWeb) is contacted to get a workunit (steps *1* and *1b*), which contains an input data endpoint. Next, the Data Lookup Server provides the client with the locations of Data Centers on the network that are active replicas of the data (steps *2* and *2b*) . Lastly, the client contacts one or more of the data endpoints (i.e., data centers) to download the data (steps *3* and *3b*). After successful completion, the client is then able to continue and process the workunit.

The roles and network messages that enable this scenario are discussed in more detail in §4.2. Note how the network complexity differs from that of a traditional super-peer network, in that message relaying is simplified and centralized.

This makes the ADICS network very similar in nature to how early Napster worked and also reminiscent of BitTorrent trackers, which efficiently manages data replicas through a centralized entity.

In ADICS, the centralized network role that manages client requests and stores metadata about the locations of download endpoints is called the *Data Lookup Service*. Similar to a BitTorrent tracker (discussed in §2.5.4.2 and §3.5.1), it keeps track of file hash information and location endpoints. However, where a BitTorrent tracker is limited to managing a single file, the Data Lookup Service acts as a metadata repository for the entire network, similar to Napster's implementation. One other major difference between both Napster and BitTorrent and the Data Center/Data Lookup Service scheme described in this chapter is the potential for the addition of security criteria that restrict these layers to a subset of the available peers and the adherence to the concepts outlined in §3.4, as seen in §4.4.

4.2 Roles

The following roles have been identified as essential towards building the proposed network: *data lookup service*, *schedulers*, *data providers*, *data centers*, *data seeds*[1], and *worker nodes* (i.e., data consumers).

The layering of these roles in the network can best be described as three tiered. The first tier is composed of a restricted number of (likely static) entities that typically would be managed by the project administrators. This fixed layer is composed of any *data providers*, the *data lookup service*, and/or *schedulers*. The next tier is the core data-sharing overlay that hosts the *data seeds* and *data centers*. The purpose of the second tier is to dynamically organize in a way that can respond to network demand. As the data-sharing layer is larger than the core management layer and also requires a larger pool of resources (in order to distribute files), it is likely that it would employ looser membership criteria. The last tier is that of the end-recipients, or *worker-nodes*, which do not provide data to the network, but rather play the important role as primary data consumers. Due to the nature of Desktop Grids, the size of the worker node network is expected to be orders of magnitude larger than the other layers, and will be responsible for

[1] Data Seeds are a specialized use-case of the Data Center role that allows for propagation of data from third party entities to the Data Center overlay. Their inclusion is needed as an integration step for the EDGeS and EDGI projects, where clients can be end-users that lack the ADICS network publication tools.

issuing the majority of *pull* requests from both the first and second tiers to locate and retrieve data.

The following gives an overview of the aforementioned network roles and their relative responsibilities and interactions in the network.

The **Data Lookup Service** (DLS) is the centralized[2] entity in the ADICS architecture that allows other network participants to locate data on the network, create or modify metadata, and publish new information. It's key roles are as follows:

- Provides a storage cache of addresses and objects
- Establishes a bootstrap endpoint to locate other network entities
- Manages authorization and authentication of publishers and retrievers of data
- Acts as a "gatekeeper" to the network by potentially restricting metadata (about data and endpoints) to authorized peers

The **Scheduler** is another centralized network entity that responds to replication requests and ensures that data are being propagated to the Data Center layer. Although a conceptually different role than that of the Data Lookup Service, in ADICS, the Scheduler role (and code) has been integrated into the DLS for simplicity. The messages that are exchanged to query and register with the Scheduler are, however, independent, enabling it to be separated for efficiency or refactoring if the need arose.

Figure 4.2 shows how Data Centers contact the scheduler on a predetermined interval (e.g., every 60 minutes) to request new work. Control of data propagation rests with a centralized entity (the Scheduler), while the burden of initiating requests lies with individual nodes in the Data Center layer. This model decouples data propagation and does not rely upon a well-constructed Data Center overlay to function, since those that wish to participate will contact the Scheduler, and those that do not, will refrain. This limits the effect of "bad" Data Center hosts, which, in a super-peer network, might not be relaying distribution requests properly.

The key functions of the Scheduler are to:

[2]The Data Lookup Service could be distributed into a DHT or other load-balancing metadata catalog if needed. However, for the purposes of this research it is restricted to a centralized entity. Even in the case of distribution, it would still serve as a conceptual "centralized" layer for metadata handling.

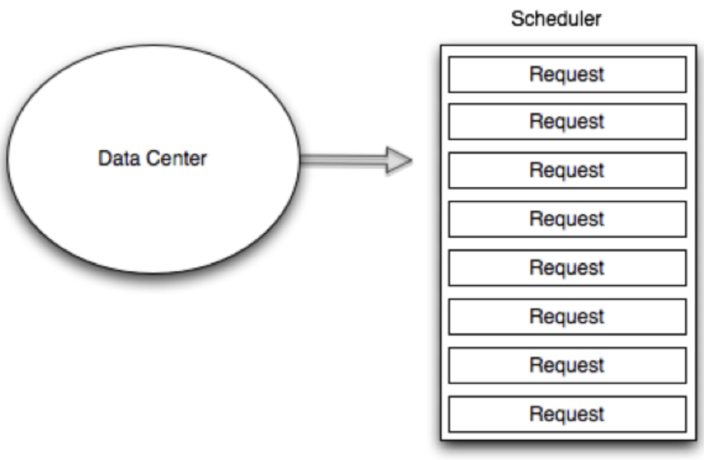


Figure 4.2: ADICS: Data Center and Scheduler Interaction — shows how Data Centers periodically contact the Scheduler to request more work in a `PULL` model.

- Receive publication requests from the DLS, specifying distribution-specific metadata such as the Time To Live (TTL) and Replication Count
- Respond to queries from members of the Data Center layer with replication objects, keeping track of the replication count to ensure that files are distributed only to their MAX level
- Tag files as stale when their TTL has expired, passing relevant DELETE requests to the Data Center layer

Data Providers are the last of the likely centralized (or limited) entities on the ADICS network. They are the peers that are able to authorize to the Data Lookup Service and publish data that will be transferred to the Data Center overlay. To share a file on the network, a Data Provider must first contact the Data Lookup Service with a publication request, providing relevant metadata, including the location of an initial *seed* for the data (see Data Seed). If the publication is successful, the metadata will be registered with the DLS and sent to the Scheduler, and shortly the Data Center overlay will begin its caching process, as follows:

- Publishes files to the network, providing needed metadata, file contents, or seed endpoint
- Returns GUID to end-user, which can be used to reference ADICS-published data

Data Centers provide the distributed data management layer that serves client requests and offloads network bandwidth and storage demands. Unlike in a super-peer network, each Data Center is an independent entity that is not linked to other Data Centers and does not contain a “neighbor list” to create the network overlay. Rather, Data Centers independently contact the Data Lookup Service (as all clients do) to receive download locations. This simplification eliminates the burden of network maintenance and reconstruction due to churn. By pushing responsibility to the DLS, authorization and authentication are also greatly simplified, at the cost of the creation of a centralized point of failure.

When a Data Center is retrieving a file from the network, it can use the fact that files are split into multiple distinct chunks to download from several other Data Center endpoints at the same time, in a mechanism known as *file swarming*, as shown in Figures 3.2 and 3.4. In ideal conditions, file swarming allows download bandwidth to be fully utilized, while exploiting multiple network paths. This enables faster downloads while concurrently protecting against corrupt data, as smaller segments of the file can be individually verified and accepted/discarded.

The key features of Data Centers are as follows:

- Store network data and acts as a partial replica of project files
- Serve requests from Workers and other Data Centers, providing file contents
- Query the Scheduler for new “work” to process in the form of file-replication requests

Data Seed nodes are special instances of DataCenters. They are able to relay requests from data publishers to add new data to the network. This is useful in cases where the data publisher might not be able to propagate initial data itself, such as when it has limited bandwidth or is restricted behind a firewall. The usefulness of data seeds becomes apparent in the next chapter when discussing integration with service grid layers, where end-users might need to publish data and would benefit from having a third-party transfer agent that could “seed” the data to the network. In addition, by providing a third-party that is able to publish data to the network, it is possible to further decentralize the authorization procedures for data publication. For example, if a DataLookup Service trusts a set of DataSeeds, it is up to each of these DataSeeds to determine on whose behalf they are willing to publish data. By decoupling the authorization needed for CRUD operations related to ADICS, responsibility can be pushed closer to the Service Grids on which they are issued. This allows for each Service Grid to implement its own policy for ADICS publications (for example, allowing all members of a certain VO to publish to ADICS, or having the job managers auto-publish), without requiring the ADICS network to be updated with each individual policy update.

Therefore, the DataSeed provides the following capabilities:

- Receives requests from clients to publish data to the network
- Authorizes to the DataLookup Server and relays publication requests
- Can serve initial copies of input data to the ADICS network (acting as an initial DataCenter)
- Potentially authorizes and maintains ACL lists of authorized users for CRUD operations to network

Worker Nodes do not perform a data sharing function in the ADICS network; rather, they are included here as the vital end-user component. The term "worker node" is used here to describe data consumers on the network that retrieve information (i.e., data), yet do not share it subsequently or otherwise participate in ADICS. ADICS worker nodes correspond to the ADICS-enabled worker nodes that would exist in a Desktop Grid environment, which could download data from the P2P network before processing a distributed job.

The functions of a Worker Node are as follows:

- Queries the network manager (e.g., BOINC or XtremWeb job distribution server) for new work
- Upon receiving new work, contacts the DataLookup Service for each ADICS input file to retrieve a list of DataCenters with the needed input files
- Requests and downloads input files from one or more DataCenters
- Works/processes workunits
- Uploads resulting output data to network manager (e.g., BOINC or XtremWeb upload server)

4.3 Data Sharing Protocol

To enable the interaction between ADICS network entities, a message protocol is needed. To aide in further discussion of how the network operates, this section introduces some of the core components of the ADICS protocol. The protocol described here is the foundation for the one used in Chapter 5 for the EDGeS/EDGI implementation of ADICS. There, a more detailed description of the messages, parameters, and interactions is given.

Here, the base messages needed for further discussion of the interactions defined in §4.2 are specified. Each message satisfies a key function needed to enable the core interactions between ADICS network entities. Note that the messages and object specifications are flexible, and can be enhanced and augmented for additional functionality and future requirements.

A `DataUnit` specifies a piece of data that is shared on the network. It has parameters such as *ID*, *Size*, *Name*, *Date*, *Description*, *MD5* sums, as well as payload information (i.e., the actual data).

A `DataPointer` is a collection of `DataUnits`, with the augmented information as to which locations on the network (i.e., DataCenters) are potential cachers.

A `DataQuery` is used to query a `DataLookup Service` for the location of data on the network as identified by a specified *ID*. A successful `DataQuery` will return a `DataPointer` that has the locations of several potential replicas.

A `DataRequest` is a request sent from an interested party to a DataCenter with a request for a particular segment of data. Note that the sequence for downloading from a DataCenter is the same regardless of the consumer, whether it is an end-user Worker Node or another DataCenter.

4.3.1 Publishing and Replication

Figure 4.3 shows how a file would be published to the ADICS network if network flooding, neighbor lists, and a distributed DLS layer were used to propagate information. In the given example, a DataCenter first instantiates a *DataPropigator* agent. The DataPropigator will hash the file and use the resulting checksum to locate the DataLookup Service that is responsible for the file's address space. The

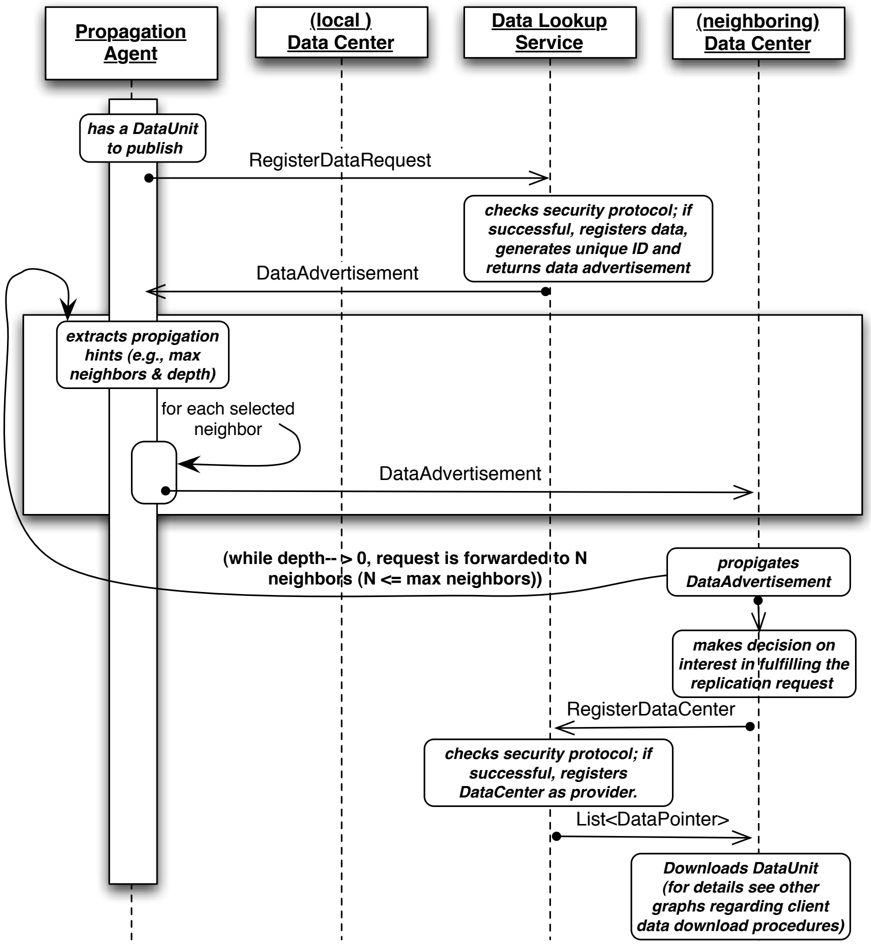

Propagation Agent
(local) Data Center
Data Lookup Service
(neighboring) Data Center
has a DataUnit to publish
RegisterDataRequest
checks security protocol; if successful, registers data, generates unique ID and returns data advertisement
DataAdvertisement
extracts propigation hints (e.g., max neighbors & depth)
for each selected neighbor
DataAdvertisement
(while depth-- > 0, request is forwarded to N neighbors (N <= max neighbors))
propigates DataAdvertisement
makes decision on interest in fulfilling the replication request
RegisterDataCenter
checks security protocol; if successful, registers DataCenter as provider.
List<DataPointer>
Downloads DataUnit (for details see other graphs regarding client data download procedures)

Figure 4.3: Publishing a File Using Network Flooding — gives a sequence diagram showing how flow of messages would look if one were to use network flooding to publish files to the network.

DLS is contacted and a *RegisterDataRequest*, which contains metadata about the file, such as maximum replications, file size, and MD5 checksums, is made. Provided that the DataPropigator has sufficient authority to publish files on the network, the DLS will respond with a unique file identifier. Note that the DLS would then potentially need to update its content network with the new information, depending on the DHT configuration.

The DataPropigator uses this ID to create a *DataAdvertisement* that is sent to other DataCenters through network flooding.[3] Contained within the DataAdvertisement is the location of the DLS that is responsible for the file. When a DataCenter receives the request, it is first forwarded onto its neighbors, then a

[3]Flooding is performed on a known number of neighbors (N) and to a certain network depth (D), which guarantees that a maximum of N^D nodes will receive the request

local decision is made on whether or not to cache the given file. If a positive decision is reached, the new DataCenter will contact the DLS, authenticate, and request to be a DataCenter. Provided authentication is successful and more replicas are needed, the DataCenter will be added to the DLS's list and will proceed with the file download.

Such a system was originally conceived for ADICS and it requires a relatively complex P2P network, with multiple levels of updating, and potentially severe performance hits if there is large network churn. Instead of the solution presented in Figure 4.3, ADICS uses a system that is much simpler and more streamlined. First, to publish a file, a Publisher (e.g., DataSeed) contacts the DLS and authenticates. If successful, it is registered as the first cache and the file replication information (e.g., TTL, max replicas, size) is transferred to the Scheduler. As the Scheduler periodically gets requests from DataCenters who are willing to cache information, it simply assigns the given file as new queries arrive. Each new DataCenter then independently downloads the file from existing network resources and registers with the DLS as a new cache. Overall, this is a much less complex distribution algorithm, which has the added benefit of transforming the replication workflow from PUSHing requests to a PULL model.

4.3.2 Downloading

Figure 4.4 gives a general overview of the message interactions that take place when a client queries for data on the network. Briefly, each *DataUnit* is described by a *DataQuery* that gives *hints* as to how to locate the data on the network. In the case of the current system implementation, which is working to locate BOINC-like data with defined data mappings, the "hint" is actually a unique ID of the data in question. However, such a strict mapping is not enforced and queries are extendable and can be application-specific, allowing for more advanced querying mechanisms and fuzzy matching. The DataLookup Service is responsible for mapping the requests to resulting DataPointer responses. It is also in charge of assigning unique IDs when files are published to the network and keeping track of the DataCenters which have registered themselves as keeping replicas of a given file.

When a client machine wishes to download a file described by a DataQuery, the following workflow takes place:

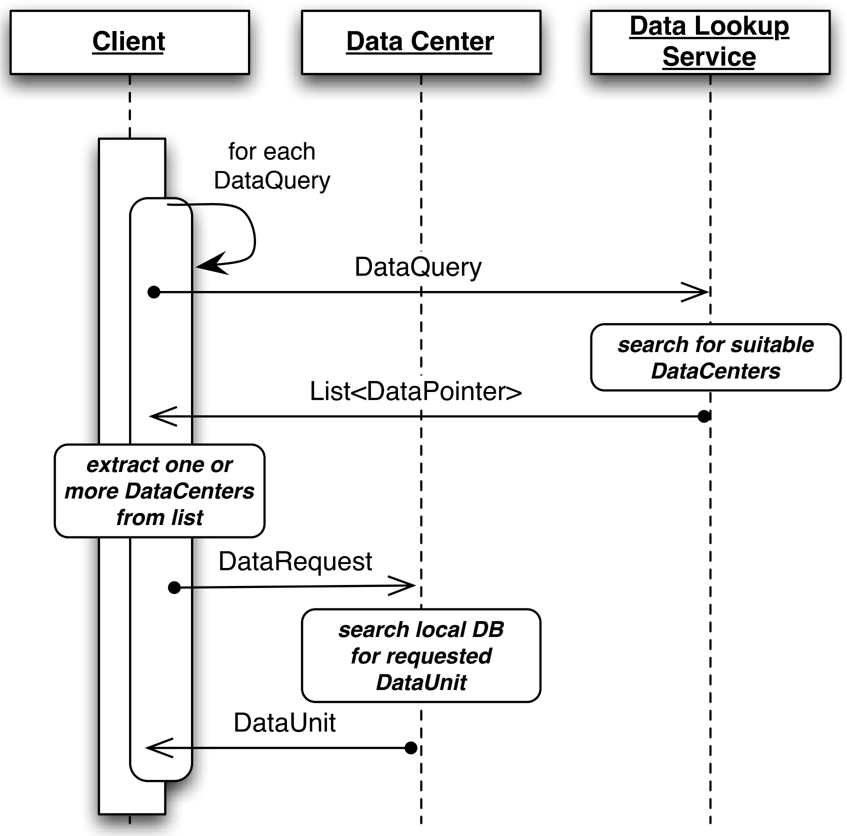

Client
Data Center
Data Lookup Service
for each DataQuery
DataQuery
search for suitable DataCenters
List<DataPointer>
extract one or more DataCenters from list
DataRequest
search local DB for requested DataUnit
DataUnit

Figure 4.4: ADICS: Flow of Messages for a Download Operation — shows the flow of messages from the client to the various other network entities in a basic data query and download operation.

1. The client contacts a *DataLookup Service* on the network and provides it with a *DataQuery* request. The DataLookup Service checks to see if it finds a match for this DataUnit. In case no match is found, it returns a 404. When a DataUnit match is found, its associated list of DataCenters is retrieved. These two properties (DataUnit metadata and List<DataCenter>) are combined into a List<DataPointer> that is returned to the client.

2. Upon receipt of the *List<DataPointer>* for a particular DataQuery, the client can extract the DataCenter location information from each DataPointer. The client is then able to directly use this reference to contact one or more DataCenters for full or partial retrieval.

3. The client generates a *DataRequest* based upon the file it is interested in receiving and sends it to a DataCenter. The DataRequest is able to indicate the specific byte-range that the client is interested in. This allows for file

swarming to take place in the event that multiple DataCenters are able to provide different partitions of the file simultaneously.

For file security and integrity checking, the DataLookup Service can (optionally) store a list of MD5 checksums for different defined chunks of the file. These allow both end-users and DataCenters to individually verify smaller portions of the file before attempting to reassemble. Naturally, the final MD5 checksum of the entire file is also supplied. For the highest level of security, it is advised that clients adhere to the chunk-size hint that is provided with the DataPointer; however, this is not obligatory. Making this an optional parameter allows for the partial and resumed retrieval of files from different DataCenters and offers the largest amount of flexibility for optimizing network bandwidth.

4.4 Evaluation Against Requirements

It is important to measure the design of a new data network for Desktop Grids against how it is able to satisfy the specific needs of the target communities and fit into a volunteer computing paradigm. In the following section, the four target issues identified in §3.4 are discussed, with emphasis on how they relate to the proposed ADICS network.

4.4.1 Scalability and Network Topology

Similar to the mechanisms employed by BitTorrent and the Julia Content Distribution Network [104], network proximity would have to be determined to map nodes adequately and decide if any are on a local network. However, if the network parameters are set to limit the participants to known hosts, the likelihood of internal LAN nodes being available to a given peer as a data center is significantly diminished. In these cases, a two-tier system of data servers is envisioned: one, in the traditional case, which meets certain selection criteria, but is available on the larger network via a public address; another which has also met the selection criteria for a “trusted node,” yet is unavailable to the larger network, but still is available to distribute files to local peers. Alternatively, LAN data centers could have lower security requirements placed upon them because the data is digitally signed to verify integrity. However, this could allow for malicious exploits involving the reporting of false results, should multiple recipients on the same LAN be given identical tasks to compute.

For cases in which the Distribution Factor is low, the use of DataCenters still might be desirable. This is because the DataCenters can act as a distributed buffer for the network, thereby providing a more consistent throughput for the nodes of the network. In volunteer systems, like BOINC, users connect at random times, which cannot be regulated by the main server. Therefore, it is likely that there are times in which multiple workers will demand data at the same time, which could be followed by periods of little activity. Farming out the data to a set of decentralized data servers could enable a buffer that would dampen down the effects of such spurious activity requests to the main server. In this scheme, the main server would push the data to the data centers consistently so that data was always available to feed the workers in the system. Then in periods of high activity, the main server would simply act as a coordinator for the downloads (i.e., tell a worker to go to data center *x* to download the data, rather than serving it itself). Such buffering would provide tolerance to inconsistencies in the network usage.

For decentralized data servers, the use of a push or a pull model very much depends on the Distribution Factor for the network. For high values of Distribution Factor, it is likely that a pull model would benefit, because many nodes will be requesting the same data and therefore the data would naturally replicate over the data centers in the network through their demand. Further, for higher values of Distribution Factor, the network may benefit by splitting data files across the data centers, using a similar scheme to BitTorrent and making the main BOINC server operate in much the same way as a BitTorrent tracker. However, data caching would be scoped to only the data centers, rather than every peer in the network being forced to replicate and serve data. The level of file swarming would need to be determined by the size of the Distribution Factor.

The proposed ADICS system uses a Napster-style "known peers" for WAN discovery, coordinated through a centralized DataLookup Service. This allows for a simplified network topology (as opposed to network flooding and neighbor lists). The drawback of such an approach would be if the centralized system fails or is unable to scale. However, as shown by the real-world Napster and BitTorrent cases, centralized metadata servers can be extremely effective and scale to large degrees, so long as they are not also burdened with core data-sharing duties. The DataLookup / Data Center partition in the ADICS network decouples these responsibilities and should allow the DataLookup Server to scale independently to meet network demand. If network size increases beyond that which a single server can satisfy, solutions such as the Julia Content Distribution Network can be explored for more advanced network topology exploitation.

4.4.2 Data Integrity and Security

Decisions as to which participants, if any, are allowed to host and redistribute data are made during the registration and replication request phases, allowing project-specific policies to govern and restrict the set of peers that are allowed to propagate data. The ADICS implementation, dubbed Attic (see Chapter 5), provides the core tools to facilitate these interactions, as well as a default implementation (using passwords and X.509 certificates). These generic tools can be extended to more complex scenarios that go beyond simple “yes/no” restricting of data center membership. For example, constraints could be introduced to govern the relative sensitivity of data and retention policies and match these against authorization tables that map certificate distinguished names (DNs) against network privileges. Adding these new types of functionalities would allow for more advanced use-cases, albeit with the additional costs of software and network complexity.

Such grouping can be used to restrict data sharing on the network to a subset of peers that match certain performance and security thresholds. By implementing an opt-in and, for the moment central, validation system for data sharing, many of the security considerations explored in [102], such as router configuration, automatically opening ports, and rouge hosts providing data, can be marginalized. In this scheme, the *data center* subset of peers on the network act as “true peers” in the sense that both send and receive on an equal standing with their data center neighbors; however, they act solely as servers to the data consuming *worker* nodes. One benefit of this approach is that *workers* continue to operate relatively unchanged from their previous working conditions, with the relatively minor addition of a distributed data lookup. Based upon the preliminary simulation results of [96, 105, 106], it is our belief that decentralized data centers can prove to be both valid and useful solutions to distributing data in Desktop Grid environments. There is, however, a tradeoff between functionality and complexity that needs to be adequately addressed and balanced if such technologies are to be adopted by production environments such as BOINC.

Depending on an individual projects configuration, firewall and router issues could be a potential problem or a complete non-issue. In a free-for-all system where any member node is permitted to be a data center, there could obviously be problems with that node's being behind a NAT. In this instance, the tradeoff between "punching holes" in the firewall and the potential benefit of the node's available network bandwidth would have to be determined. For more restricted systems, in which pre-specified static or semi-dynamic nodes are dynamically promoted to be data centers as the network requires, firewall and router issues could be minimized, for example, through enforcing eligibility criteria for data centers to only those nodes that have a publicly addressable network space. In this instance, *semi-dynamic*, is referring to nodes that have gone through some pre-screening that verifies them as good candidates for data-centers, such as obtaining a specific certificate or accumulated substantial project credits. However, when they actually perform as data centers is determined dynamically, based upon network properties.

Current design of ADICS is working with the assumption that a more secured sharing will be desired and enforced. This requires data center peers to be publicly accessible machines, thereby for the moment forgoing the potential pitfalls of attempting to implement automatic firewall configuration, leaving this as a future implementation issue outside of the scope of this research.

4.4.3 User Security and Client-Side Configuration

As with *Data Integrity and Security*, the issue of how much relative freedom network participants have to manipulate the network will depend on the individual policies of each hosting project. In the most restrictive case, the only nodes that would be allowed to propagate data would be well known and trusted, thereby affording the same level of security currently available in the centralized network. In looser security configurations, which are configured to harvest more participant network resources, the security issues would be roughly equivalent to BitTorrent, as discussed in [95]. The advantage of the system proposed here is that there are middle-ground options lying between these two extreme alternatives that could be exploited.

ADICS relies on the data signing and validation procedures currently utilized by BOINC, which essentially guarantee that requested data will be what is ultimately retrieved. However, to effectively distribute a single data file from multiple data centers to an individual host, BitTorrent-style file-swarming techniques are being investigated. This requires two-level hashing of data, once on the individual chunks and once on the entire file. This additional chunk-level hashing helps prevent malicious or misconfigured DataCenters from propagating “bad chunks” to the network.

Authentication is the verification process by which an entity identifies itself to others and gives evidence to its validity. Public key infrastructure is a proven tool that can be fairly effectively applied for performing peer identity authentication. In the simplest case, this can be done by having a central authority (i.e., the BOINC manager) sign and issue either full or proxy certificates to those it deems trustworthy enough to distribute data on its behalf.

When another peer on the network contacts this “trusted” entity, it can use the public key from the centralized BOINC manager to verify the authenticity of the trusted peer. This process can likewise be performed in reverse, provided clients also are issued certificates, as a means for the data distributers to validate the identity of the clients and verify that they have the proper credentials to retrieve data. The process of using certificates for mutual authentication can be a fairly effective solution that would provide individual peers with certainty that the host they are retrieving data has been delegated the proper authority and *visa versa*. More interesting use-cases that provide for interaction between multiple virtual organizations (VOs) and hierarchal delegation (e.g., certificate-chaining and cross-certification agreements) can be derived from this simple arrangement, but are beyond the scope of this research [92, 93].

For the BOINC environment, possible ways in which a reasonable level of authorization could be implemented would be to tie certificate issue into the centralized scoring system that keeps track of users and groups that are contributing cycles to the project. In such a scenario, when a new potential data server enters the network, it would contact the group of data servers in the network and offer to join them. After certificate exchange to authenticate identity, a lookup could be done against the scoring system to see if the newcomer meets the requirements to join the data center network. If this is successful, the newcomer could then be added to a centralized or decentralized list of “authorized data centers” or alternatively added to the table of contactable centers from the other centers.

If this were implemented in the absence of a centralized authority, one issue that would arise would be how to continuously validate the data center layer to remove data centers that have turned rouge or been compromised since joining the network and passing the initial validation step. One potential solution would be to issue only proxy certificates that expire after a given amount of time, thereby requiring periodic re-authorization. Then when clients connect to a stale data center, they could verify the lifetime and expiration date of the certificate.

For the proposed ADICS network, the authorization and authentication can be further simplified, since the DataLookup Service essentially controls the access lists of what agents are able to publish or retrieve information from the network. Similarly, the DataSeeds can maintain tables of authorized “sub publishers” that can use them as network proxies. By choosing the network design proposed previously, ADICS is able to offer simplified and easy-to-maintain network authorization and role management, while restricting entity membership and limiting access to network resources.

4.4.4 Legacy Software Integration

The (potentially, depending on project setup) “secure” DataCenter approaches outlined in this Chapter, in the form of ADICS, would demand radical changes to existing software such as BitTorrent or even generic P2P network-building middleware like JXTA. This is primarily due to two distinct areas: internal integration with Desktop Grids and external library dependencies. Regarding internal integration, BitTorrent or any other solution beyond FTP or HTTP would require changes to the core BOINC client code to handle the new protocol. This is problematic, as there are tens of thousands of BOINC clients that would then need to be updated in order to enable P2P file sharing. Moreover, the restriction to allow opting-out of the system makes it difficult to apply BitTorrent-like solutions which enforce participation.

ADICS does not require workers to participate in the data distribution. Therefore, the fundamental software integration issue that confronts ADICS is how to allow BOINC (and other Desktop Grid software) to use the ADICS network without requiring changes to the core client code. A solution to this problem is presented in the next chapter (see §5.3), where a proxy project is run as a daemon in the background and intercepts P2P network requests. This solution allows non-intrusive legacy software integration.

4.5 Validation of Design through Experimentation

After the initial ideas for the ADICS network were conceptualized, basic simulations were performed using NS-2 [107, 108], to evaluate how packets would flow given the ADICS network topology and identify potential bottlenecks. Next, a generalized "dynamic caching" data distribution paradigm was explored in collaboration with ICAR-CNR using a custom simulation framework [99, 105]. The work presented there was more generalized than the ADICS network that has been proposed here; however, the results and arguments for P2P data distribution led to a refinement of the ADICS protocol. Specifically, it presented an argument that using dynamic data caching, while knowing the network and data properties, allows for a more efficient configuration of data server replication, as opposed to the current static-sized set used by BOINC projects. During these initial simulations, much was also learned about how the network would be able to scale and what proportion of network resources would need to be partitioned as data providers to adequately satisfy demand.

Based upon the preliminary results of [105], the ADICS protocol was further developed [85, 99]. This section introduces the subsequent simulation work[4] in the development of ADICS, which was used to test its validity in a Desktop Grid environment and verify the design before implementation. It should be noted that the simulations here use realistic BOINC project metrics. This allows not only for the validation of the ADICS network architecture, but also helps to fine-tune run-time environment parameters, such as Data Center loads and network topologies. For a discussion of further testing and results of the protocol running on a deployed Grid, please see §6.2.

Recall Figure 4.4, shown earlier in this chapter, and repeated here as Figure 4.5. It shows the basic entities that will be simulated for the ADICS network, as well as the sequence of messages exchanged in the system. There are several kinds of nodes, each of which can implement one or more different *roles*. The following gives a brief review of the different roles and the associated functionalities, as they pertain to the simulations given in this section:[5]

[4]The simulations introduced in this section were done in collaboration with colleagues from the University of Calabria and ICAR-CNR (specifically, Daniela Barbalace and Carlo Mastroianni), who designed the simulation software presented here and have been instrumental in providing this analysis.

[5]Note that the functionalities for the simulations are simplified versions of those in the ADICS protocol and may differ slightly from those presented in the protocol specification.

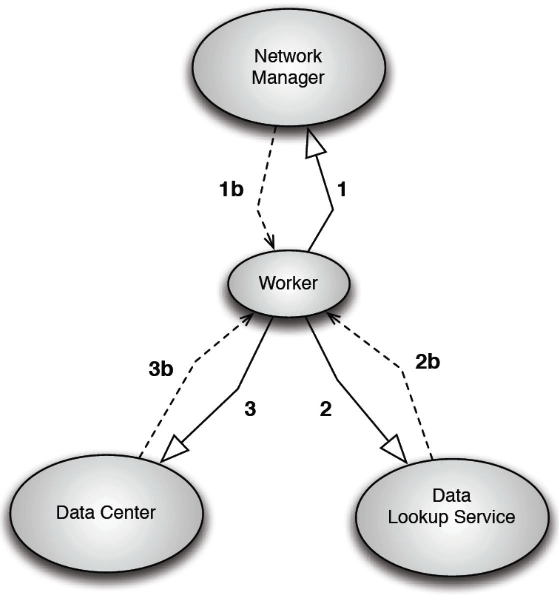


Figure 4.5: Client Message Workflow Used for Simulations — shows the basic flow of messages from a Worker to the various other network entities used for ADICS simulations.

- The *Network Manager* manages the *Worker Node* requests, assigning them the workunits (WUs) that represent distributed and computationally intense simulations to be run on the network. The Network Manager stores a copy of all the input data and, at the beginning of data transfer, it distributes these items to the *Data Center* overlay, which acts as the caching layer serving download requests from *Worker Nodes*. The DLS retains an up-to-date cache of *Data Center* addresses and the list of items each has stored.

- The *Data Center* receives some data items from the Network Manager and stores them for *Worker Node* requests. When a worker asks a cacher for some data, it sends this data through a direct connection.

- The *Data Lookup Service* (DLS) is a node that plays this role and should provide the worker with a list of *Data Centers* that store the data item needed to execute an assigned workunit. After contacting the DLS for location information, *Worker Nodes* can contact the *Data Centers* (i.e., cachers) directly to retrieve the data item. The exact procedure to retrieve data is described later in this section. For the purposes of the simulations provided here, it is assumed that the Data Lookup Service is included in the Network Manager role, but in general, these nodes could be different.

- The *Worker Node* is a node able to execute a workunit. It periodically

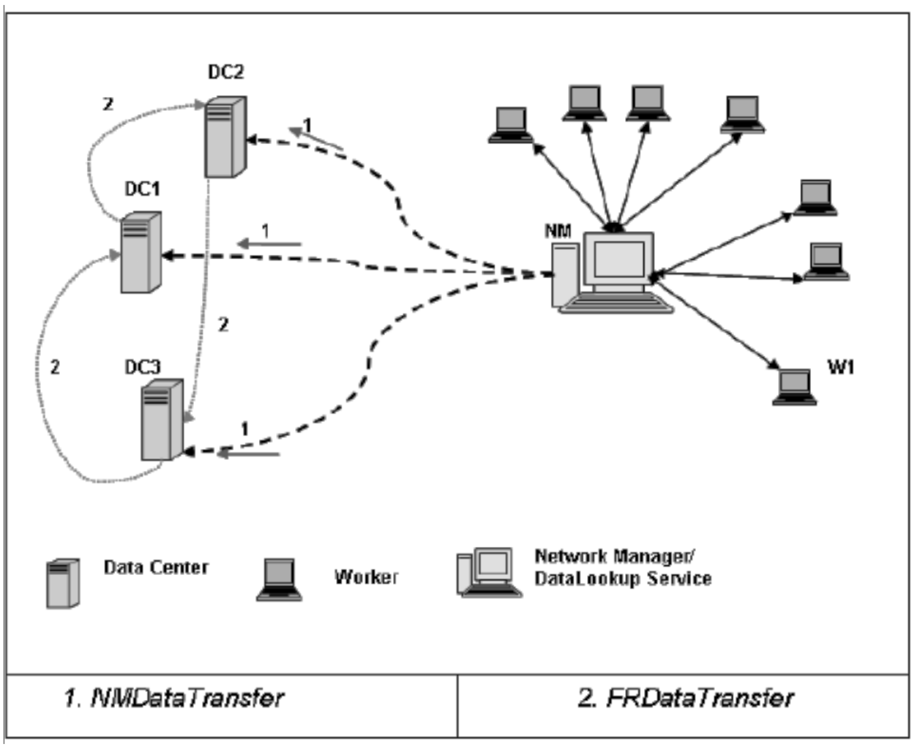


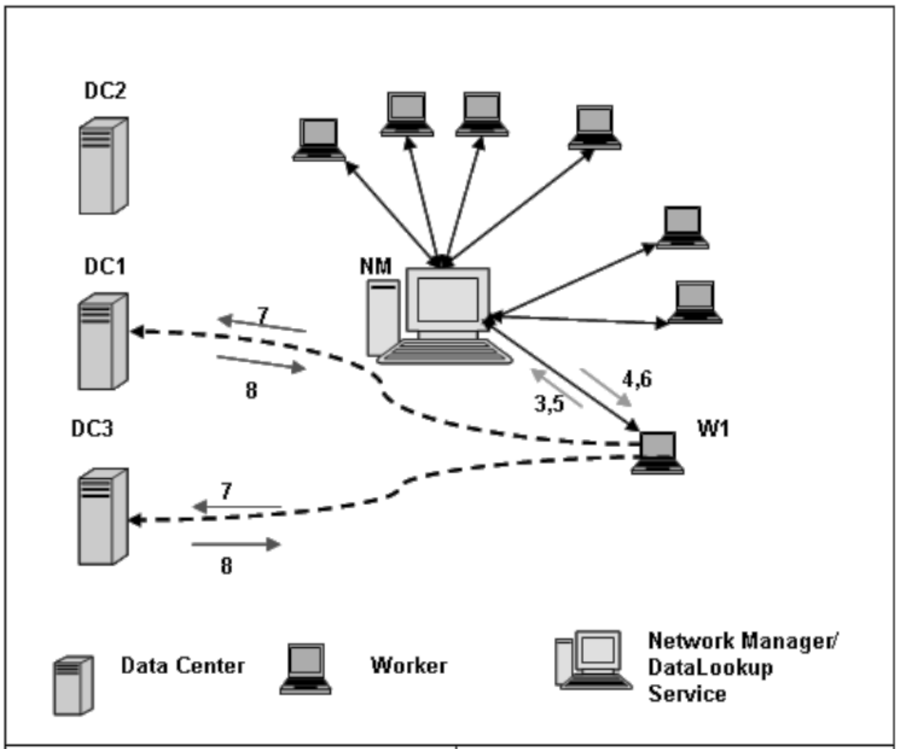
Figure 4.6: Caching Algorithm in a Sample Network — shows the flow of messages when caching a file in the Data Center overlay.

queries the *Network Manger* for a new job and subsequently retrieves required input data items needed for job execution. When it finishes processing a workunit, each Worker Node produces some results. For the purposes of the simulations performed here, the results of workunits, and their associated output data, are ignored, due to their inconsequence in measuring input data distribution.

Figures 4.6 and 4.7 depict the sequence of messages exchanged among the Network Manager, the Data Centers, and the workers. In particular, this sample network contains three Data Centers, seven workers and the Network Manager (which includes the functionality of the Data Lookup Service).

At the beginning, the Network Manager sends its data items to the cachers (i.e., Data Centers). The particular caching strategy adopted in this work is the one described in [90]. The aim of the simulations here and the initial phase of input data deployment in ADICS is to support a FastReplica method, which has shown to be efficient and reliable at quickly replicating large files.

There are a few basic ideas exploited in FastReplica. In order to replicate a large file among n nodes, the original file is partitioned into n subfiles of equal



3. GetWorkUnit	*6. DataLookup Response*
4. WorkUnit Response	*7. Data Query*
5. DataLookup Request	*8. Data fragment*

Figure 4.7: Workunit's Execution in a Sample Network — shows the flow of messages when a client downloads the input files for a workunit.

size. Each subfile is transferred to a different node in the Data Center group. After initial distribution, each node propagates its subfile to the remaining nodes in the group. Thus, instead of replicating the entire file to n nodes using n Internet paths, FastReplica is able to exploit $n * n$ Internet paths within the replication group, where each path is used for transferring only $1/nth$ of the file. This also avoids the bandwidth of the unique data source (e.g., the Network Manager) from becoming saturated, since the bulk of the transfer is happening within the data caching overlay.

In the simulation work presented here, as in FastReplica, input files are divided among cachers. Data files are represented as a set of data items, with each data item corresponding to a subfile that is cached in its entirety. Moreover, a new parameter, *DReplicationFactor*, is introduced that indicates the number of Data Centers on which each data item should be replicated. This is done to limit bloating of the network and allow partial caching, since it is not useful to replicate all files (in their entirety) to all cachers on the network. Rather, partial replicas are useful to distribute load and provide an acceptable quality of service, allowing, for example, a 1 GB file to be split into one hundred 10 MB subfiles (*aka* chunks) and replicated to multiple points on the network. The DReplicationFactor specifies how often each of these “chunks” should be replicated, providing a distributed replica of the file among a large number of cachers.

For example, as shown in Figure 4.6, notice that there are three data items and a DReplicationFactor of two. Therefore the Network Manager sends a data item to each cacher (step 1, NMDataTransfer), and the cacher propagates its data item to another cacher (step 2, FRDataTransfer), in order to replicate the data item to two nodes (as requested by the DReplicationFactor value of two). Thus, each cacher sends the item to $DReplicationFactor - 1$ other randomly chosen data cachers.

Each of the scenarios presented here has a total number of workunits (aka jobs), *NJob*, and a number of different data items, *NDataItem*. The same data item can be given as input for different jobs. The number of times a data item needs to be assigned to workers is on average: $NdataAssign = Njob/NDataItem$. Moreover, there are N data cachers (*Ndatacachers*) and each data item is replicated on DReplicationFactor of these. In the absence of swarming, DReplicationFactor should be lower than NdataAssign, so that each of these DReplicationFactor data cachers can serve several job requests.

FastReplica helps to mitigate the scenario when the Network Manager becomes a bottleneck. Instead of sending to the network a number of data items equal to (NDataItem) multiplied by the replication factor (DReplicationFactor), FastReplica transfers each data item (NDataItem) once, and subsequently each cacher is responsible for the completion of the replication algorithm to the other caching nodes.

Figure 4.7, shows the steps needed for the execution of a workunit. A Worker Node, W_1, joins the network and connects directly to the Network Manager NM, asking for a job with a GetWorkUnit request (Step 3).

In Step 4, the Network Manager answers the Worker, using a WorkUnit Response, and assigns a workunit to the Worker Node. The message also contains other information, which the Worker can use to ask the Manager for components the worker is missing, for example, the required data items.

After the assignment phase, the Worker retrieves the data item for the assigned workunit. The Worker must search for a Data Center that stores the required input data. The Network Manager keeps a cache of all Data Center locations. The Worker sends the DataLookup Request (Step 5) using the information of the previous message (WorkUnit Response), to the Data Lookup Service (for the purposes of this simulation, this is the same entity as the Network Manager). The Data Lookup Service/Network Manager sends a DataLookup Response (Step 6) back to the Worker, which contains the location of one or more Data Centers that have the needed data. In the final steps of downloading, the Worker retrieves the data item fragments simultaneously from different sources (Steps 7 and 8), by using file swarming, as in BitTorrent (see §2.5.4.2), to minimize the download time.

The message sent by the Worker to download data is a Data Query. When a Data Center receives a Data Query, it responds with a Data Fragment, which contains the requested data payload.

In the above scenario, when the Network Manager receives a query for a WU, it returns to the Worker a list of Data Centers that have that data item. The worker can download the fragments of this data item in parallel from different sources. If the number of fragments (*NFragment*) is lower than the number of Data Centers that have that data item, the Network Manager will choose $NFragment$ Data Centers among those that are less loaded to put in the list. Each data packet sent to the Worker contains the data itself (the payload) and other support information.

As mentioned previously, the simulation presented here analyzes the efficiency of input file distribution. The result sets that are generated from processing workunits are out-of-scope and not taken into account.

The phase in which the Data Centers (*aka* Data Cachers) acquire data items and the one in which Workers request these items are executed in parallel. When a Worker requests data locations from the Network Manger, it receives the most recent list of Data Center addresses, which is updated as new cachers propagate information.

To achieve the simulation of the ADICS network topology presented in §4.1, the simulation software presented in [96] was modified to support the ADICS and EDGeS project demands. In the following, the most substantial differences and modifications are listed:

- In EDGeS, the Data Center and the Network Manager are contacted directly, not through a P2P search. In the previous version of the simulator, a

P2P search was performed each time a worker had to contact the Network Manager or a Data Center for the first time. This is the fundamental difference between a network flooding approach used by many P2P software implementations and the centralized discovery system proposed in ADICS.

- In EDGeS, the Network Manager is the only super-peer in the network, to which all the Workers must be connected. Workers are not connected among themselves. To be able to contact a Date Center, each Worker must contact the Network Manager, which retains a list of available Data Centers and relays those addresses to Workers. Conversely, in the previous simulation work presented in [96] and [105], Workers could be directly connected to a Data Center or Rendezvous node for message relaying and discovery.

In order to validate the protocol using realistic network metrics, stats from three of the most important BOINC projects were gathered: Rosetta@Home, Einstein@Home and SETI@Home. Below is a list of the most relevant parameters that can be replicated through simulation for each of these projects:

- Size of a workunit (WU)
- Processing Time of a WU
- Size of Initial Data (The first WU that a worker performs requires more data, because, for example, it needs the source code of the application.)
- Results/Day. The number of results products in one day corresponds to the number of WUs correctly performed by all the workers within one day
- Participants (Worker pool size)
- Replication factor. The replication factor defines how many times a WU must be performed in order to check results coming from different workers. It must not be confused with DReplicationFactor, that represents the number of times a data item should be replicated on the cachers.
- %NewWorkers: indicates the percentage of new users in the network, that is, how many workers ask for a WU for the first time. This helps to simulate network churn.

Tables 4.1, 4.2 and 4.3 show the stats mentioned above for these three projects.

Table 4.1: Rosetta@Home Project Statistics

Size of a Workunit	3 MB
Processing Time of a Workunit	3 hours
Size of Initial Data	17 MB
Results in a Day	115,000
Number of Data Centers	variable from 3 to 20
Participants	100,000

Table 4.2: SETI@Home Project Statistics

Size of a Workunit	340 KB
Processing Time of a Workunit	2 hours
Size of Initial Data	2.5 MB
Results in a Day	1 Million
Number of Data Centers	variable from 3 to 20
Participants	500,000
Replication factor	2

For the purposes of the simulation analysis presented here, the number of Workers was reduced in order to obtain results more quickly. Consequently, the size of the projects was scaled appropriately to keep results consistent and accurate. In the remainder of this section, the performance indexes that were computed through the simulations are analyzed.

- The *percentage of bandwidth utilization*: in previous simulations, the utilization of Data Centers was calculated as the percentage of time in which the Data Centers have at least one active connection (there is, at least one Worker that is retrieving data). This metric proved not to be very useful for analyzing network demand and the efficiency of distribution. Therefore, for the simulation work shown here, it is the percentage of bandwidth used on each Data Center in a given time that is analyzed, lending the ability to gauge network throughput and demand distribution.

  In BOINC it is not realistic to assume that the bandwidth of a Data Center is equal to that of a Worker, as the Worker-node layer generally consists of a large number of edge nodes. Data Centers are assumed to be a more powerful computers with larger available bandwidths (for example, 100 Mbp/s) than the Workers, which usually have only consumer Internet connections

Table 4.3: Einstein@Home Project Statistics

Size of a Workunit	3.2 MB
Processing Time of a Workunit	5 hours
Size of Initial Data	40 MB
Results in a Day	50,000
Number of Data Centers	variable from 3 to 20
Participants	200,000
Replication factor	2

an order of magnitude less. Therefore, it is useful to determine how many connections are active in a given time span, and what percentage of a Data Center's bandwidth is used by those connections. For example, if the number of connections to a Data Center, multiplied by the bandwidth that each Worker is using (e.g., 10 Mbp/s), is lower than the total available bandwidth of the Data Center (100 Mbp/s), then each Worker will have its maximum download bandwidth available (i.e., 10 Mb/s). Alternatively, if the total available bandwidth of the Data Center isn't sufficient to satisfy all Workers at full speed, it is fairly divided among the connected Workers (resulting in speeds $<$10 Mb/s). The same strategy of bandwidth sharing and division has been implemented between the Network Manager and the cachers.

- The *average speed of download*: this index is defined as the average speed of a Worker (in Mbp/s) in downloading the data items used to process the assigned workunits.
- The *percentage of new users*: due to the vastness and volatility of volunteer computing projects, which often involve tens of thousands of computers, many of which leave the network while others join, the simulations shown here account for a continuous arrival of new Workers. This is important because of the (potential and substantial) difference between old and new Workers that occurs during the initialization stage. For example, new Workers often download more data, as in the case of the Einstein@Home project, where the first WU requires 40 MB rather than 3.2 MB (see §2.3.3.2 for explanation). To simulate old Workers leaving and new Workers joining the network (a phenomenon known as "network churn"), each simulated Worker downloads the initial data (e.g., 40 MB) not only for the first workunit, but also with each subsequent workunit given a certain probability (parameter %NewWorkers is set to 20% by default). Thus, when the new %NewWork-

ers trigger goes off, the act of an old Worker leaving and a new Worker joining is simulated, and the tests are able to account for network churn.

- The *average download time* is the average needed time to download an entire data item.
- The *overall execution time* is the time needed to retrieve and process all the workunits, that is, to complete the given simulation run.

Performance Evaluation

The simulation analysis was performed using an event-based simulator and the parameters shown in Table 4.4. The ADICS/EDGeS scenario and the related network and protocol parameters are set to assess the representative BOINC applications identified earlier in this section. The `DReplicationFactor` was fixed to three for the experiments and the size of each project was scaled (as well as result sets) to quickly run the simulations. A table containing the values of the number of Workers and of results/day is reported below.

Table 4.4: Values Used in the Simulations

Rosetta@Home	
Number of Jobs	12,000
Number of Workers	10,000
SETI@Home	
Number of Jobs	10,000
Number of Workers	5,000
Einstein@Home	
Number of Jobs	5,000
Number of Workers	10,000

Figure 4.8 shows the average bandwidth utilization vs. the number of Data Centers, for the three different projects. It can be noticed that, as the number of Data Centers increases, their bandwidth utilization decreases. This is an expected result and occurs because Workers can download the needed data from additional sources, reducing each cacher's individual load. However, this trend has an optimum: if the number of Data Centers increases over a certain threshold, the bandwidth utilization does not decrease further; this is primarily due to

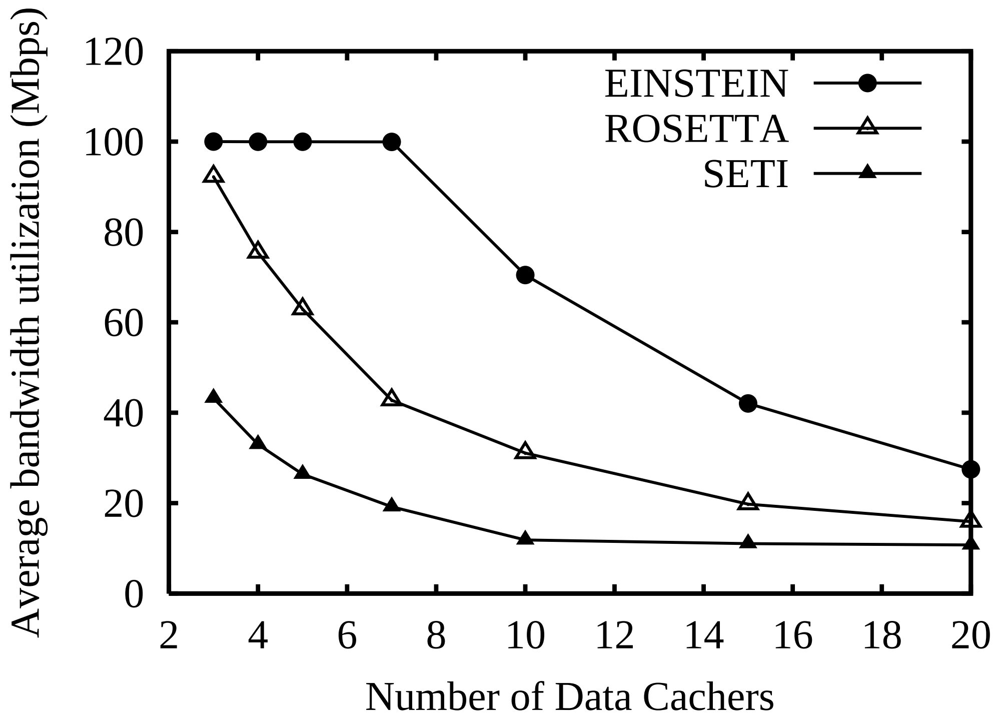


Figure 4.8: Bandwidth Utilization vs. the Number of Data Centers — shows the average bandwidth utilization vs. the number of Data Centers, for each of the three selected projects.

the new cachers not being selected as download targets, as the network already has sufficient capacity. Therefore, they sit idle.

It also can be seen in Figure 4.8 that the different size of the data items in each project impacts heavily on the optimum number of the Data Centers that should be used. For example, SETI@Home, has relatively small input data ($\approx$ 340 KB) and simulations show the optimum number of Data Centers is around 10[6]. However, for Einstein@Home, more Data Centers (i.e., 20) are required, due to the larger size of the input data items and different numbers of Worker nodes.

We can derive the optimum number of Data Centers from Figure 4.9. For each of the three different projects, the mean speed of the Workers to download data

[6]It should be noted that the "optimum" number of Data Centers is directly related to the network bandwidth metrics of the Data Center overlay. In a network with slower Data Centers, more would be required, and *visa versa*.

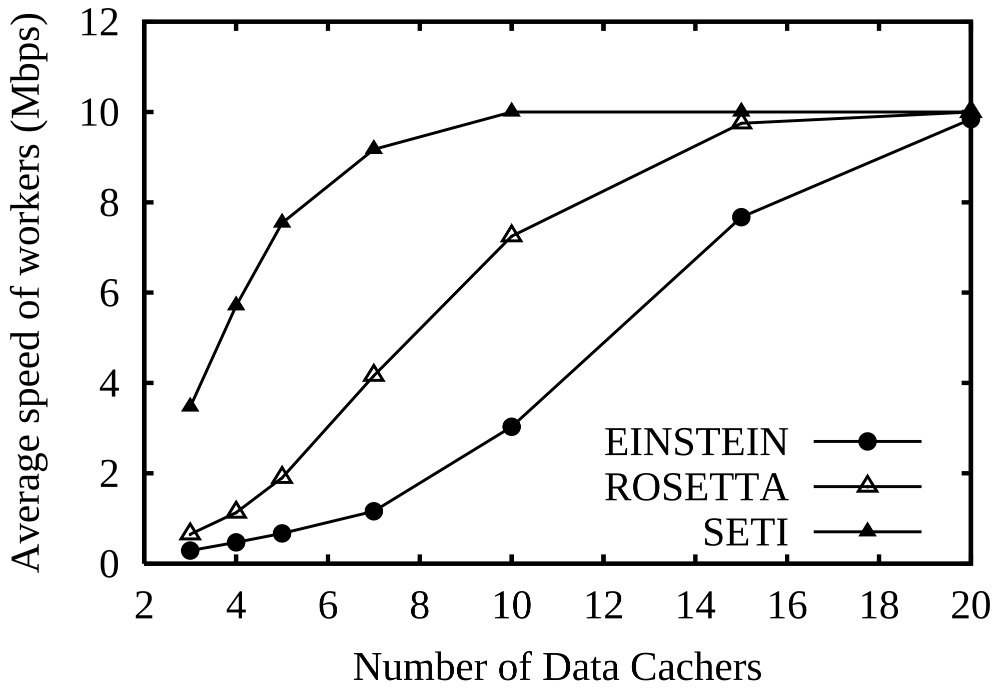


Figure 4.9: Download Speed of Workers vs. the Number of Data Centers — shows the average download speed of Workers vs. the number of Data Centers, for each of the three selected projects.

vs. the number of Data Centers is shown. Ten Data Centers allows Workers to fully utilize their download bandwidths for SETI@Home, whereas 14 are needed for Rosetta@Home and 20 for Einstein@Home.

Figure 4.10 depicts the mean time that a Worker takes to download a data item from the Data Center overlay. As the number of cachers increases, the download time decreases, but the threshold mentioned above is confirmed here also. If the number of Data Centers increases over a certain threshold, these nodes are not proportionally used and the network experiences diminishing returns when adding cachers.

Finally, in Figure 4.11, the trend for overall execution time vs. the number of Data Centers can be seen. It is interesting to notice that, for SETI@Home, this value is almost constant. This behavior could be explained by looking at the execution time of a workunit. While in Einstein@Home and Rosetta@Home the

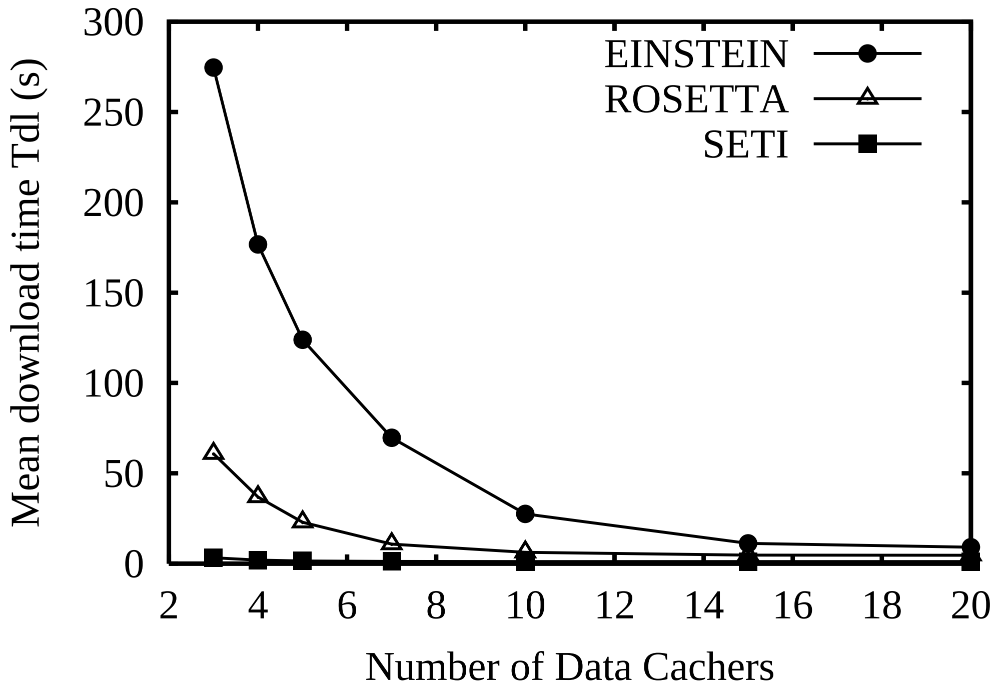


Figure 4.10: Download Time vs. the Number of Data Centers — shows the average download time vs. the number of Data Centers, for each of the three selected projects.

process time and the download time are comparable, in SETI@Home the download time is proportionally much smaller than the execution time. Therefore, for SETI@Home, an increase in the number of cachers does not have as pronounced effect on the overall time to completion as it does with the other projects.

4.6 Summary

The simulations show that the ADICS network can solve real-life volunteer computing data distribution needs with a relatively small number of Data Centers. It should be noted that in all three examples, the number of optimal cachers was greater than the current centralized mirroring employed by projects. This can be attributed to the large cost of running a data mirror for Desktop Grid projects. Since mirroring requires a full copy of all input data, rather than the partial caches

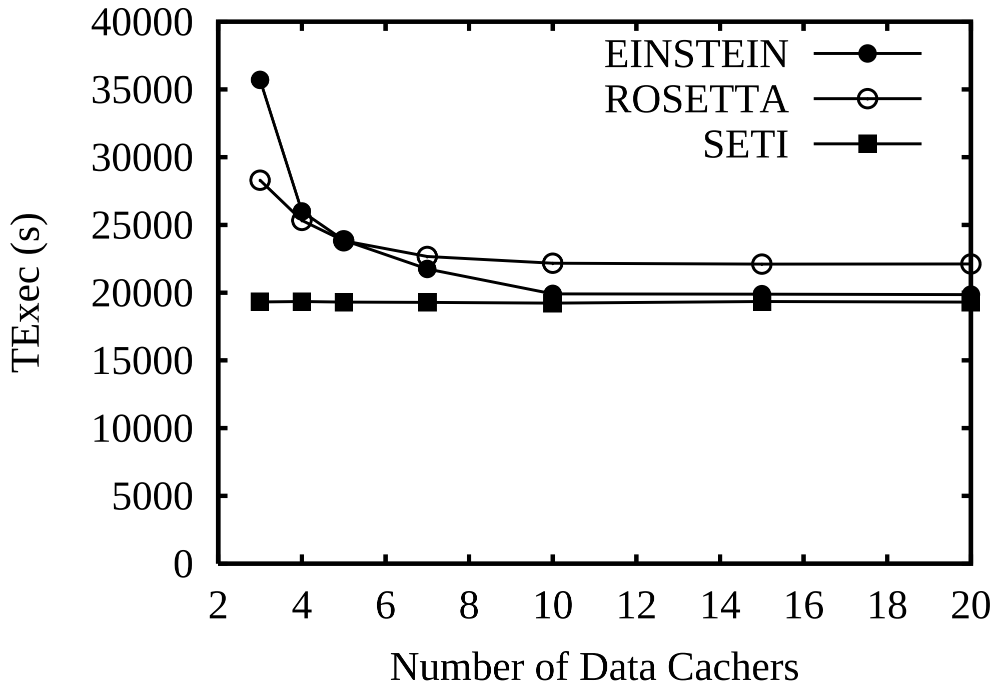


Figure 4.11: Execution Time vs. the Number of Data Centers — shows the overall execution time vs. the number of Data Centers, for each of the three selected projects.

used by ADICS, the hardware requirements (e.g., storage and bandwidth) to host a mirroring site are large. Also, due to the manual nature of setting up mirroring servers (there is no automatic system for doing so in BOINC or XtremWeb), additional system administrator time is needed to setup and maintain a data serving site. This results in fewer, very powerful and capable, mirroring locations being used to distribute project data.

ADICS provides an alternative, where the threshold to serve project data is greatly reduced, allowing a broader, yet less intensive use of caching sites to serve project data. The benefits of a more widely distributed and more highly scalable network are seen not only in automatic load balancing and increased participation, but also in a potential decrease in overall hard maintenance costs, which can inhibit volunteer computing solutions for projects that lack the dedicated infrastructure to host and serve their input data.

The software architecture introduced here was showcased in the Enabling Desktop Grids for e-Science (EDGeS) project. It was prototyped in Java and used serialized Java Objects as a messaging layer and Remote Procedure Calls (RPC) to invoke actions on network entities. ADICS was successfully prototyped in the EDGeS project and showcased in the EDGeS reviews and demonstrations. Following validation of the research ideas presented here, a follow-on project (EDGI) was created (see Appendix B), where ADICS was further developed into a more versatile and useful software infrastructure.

The ADICS architecture remained the same in EDGI, with changes taking place in the format of the way messages are relayed around the network. Instead of passing Java Objects, an XML-based REST model was implemented, allowing for better integration with legacy applications and Service Grid infrastructures. Chapter 5 of this thesis delves into the details of this second incarnation of ADICS, dubbed “Attic.” It shows the details of Attic’s implementation and how the REST-model helped to further solve the issue of data distribution within Desktop Grids. In addition, the next chapter gives details regarding the data transitioning between Service and Desktop Grid infrastructures.

CHAPTER 5

Implementation and Integration with Service and Desktop Grids

The goals of both the Enabling Desktop Grids for e-Science (EDGeS) and European Desktop Grid Initiative (EDGI) projects (see §2.4 and Appendix B) are to transition jobs from Service to Desktop Grids. EDGeS also contained a component for transitioning Desktop Grid jobs to Service Grids; however, this particular feature is not discussed in detail here as it was not included in the follow-on EDGI project. EDGI expanded upon EDGeS to move much of the prototype software researched there towards a production system. In addition to these goals, EDGI also seeks to integrate other Service Grid architectures, namely the Advanced Resource Connector (ARC) [109] and the Uniform Interface to Computing Resources (UNICORE) [110].

When discussing the research and integration that took place in EDGeS and is currently taking place in EDGI, there are three main data-related components: first, the data distribution framework that implements the design given in Chapter 4; second, the Service Grid integration steps and tools that allows publication of files to the data distribution framework and incorporation of these new inputs into Desktop Grid workunits; lastly, the client-side (legacy) application changes that needed to take place to enable Desktop Grid users (i.e., BOINC and XtremWeb clients) to retrieve the referenced input files.

This chapter investigates each of these key components in detail and gives the relevant implementation details. The combination and intersection of these three aspects provides a concrete example of the research ideas proposed in this thesis. The resulting implementation is a data-sharing network for Desktop Grids that provides a usable solution to the integration challenges of Service to Desktop Grid data migration. Its development, acceptance, and ultimate use in the community serves as a demonstration of the applicability and proof of the hypothesis developed in Chapter 1.

Acknowledgments

An undertaking and integration challenge as large and complex as that proposed in this work could not be achieved without the active cooperation and help of many experts in both the Service and Desktop Grid communities. Most of the software introduced in this chapter was developed in close cooperation and coordination with European Union partner institutions throughout both the EDGeS in EDGI projects. During the EDGeS project, Cardiff University was the lead institution for the data management research activity (JRA3). As head of JRA3, I steered the design of Attic (see §5.1) and actively contributed to its development. Likewise, in EDGI, Cardiff University is responsible for the data management activities inside both the research (JRA1) and infrastructure (SA1) work packages. Although I have been, and currently am, responsible for coordinating most of the data management development that has taken place in both EDGeS and EDGI, many others within the project deserve acknowledgement for their efforts. Here, I identify some of the key players and institutions that deserve credit for their efforts in developing the data management infrastructure showcased in this chapter.

First, the Attic design and development effort involved the minds and programming skills of many researchers and students at Cardiff University. Notably, Andrew Harrison was instrumental in key design features and the REST-based implementation of Attic that is presented in this chapter. My Ph.D. supervisor, Ian Taylor, also actively contributed to the Attic design and provided guidance in searching for distributed data management solutions. Moreover, certain features of Attic were investigated and implemented (under varying levels of guidance) as projects by other research staff and student helpers employed on both the EDGeS and EDGI projects.

Second, the migration paths from Service to Desktop Grids involved the coordinated efforts of many institutions and stakeholders. The JRA1 work package of EDGeS and also the JRA1 work package of EDGI were both instrumental in the development of the bridging technology that integrates Attic with the ARC (see §5.2.2), UNICORE (see §5.2.3) and gLite (see §5.2.1) Service Grids (for a background discussion, see §2.4). These activities were often complex and involved efforts from several different partners in the development of scenarios, the design, and the Service Grid component implementation. This list notably includes researchers at SZTAKI (gLite), University of Copenhagen (ARC), and Paddleborn (Unicore); however, others have also contributed.

The BOINC plug-ins for Attic (see §5.3) wouldn't have been possible without design help from the BOINC community and substantial programming efforts on the part of students and researchers at Cardiff, notably Kieren Evans. Likewise, without development and testing efforts from researchers at IN2P3, the protocol handlers and integration with XtremWeb (see §5.4) would not have been possible.

Software License

All software described in this chapter and provided by both the EDGeS and EDGI projects is available under open-source software licenses. The Attic infrastructure detailed here, which is the core output of this research, is released under a liberal Apache license, making it available for public, private, and commercial use.

Availability

Attic, as well as the Attic BOINC proxy project, are freely available on at the Attic website [111].[1] Attic source code is also hosted on GitHub [112] for long term sustainability. Other software components, such as the 3G Bridge, ARC, UNICORE, and gLite, are likewise available in public source-code repositories. Links to their documentation and related files can be found on the EDGI website [113].

5.1 Attic

The following section gives a description of the *Attic* architecture. Attic is the second iteration of the ADICS architecture described in Chapter 4. The first iter-

[1]The latest technical developer's documentation (e.g., Javadoc API guide), installation instructions, and software downloads are available at this location.

ation had essentially the same message protocol and network entities; however, it lacked a REST-interface and protocol handler implementations. The software was renamed “Attic” to represent more fully its functionality as a place to “store things,” as well as an opportunity to remove a long acronym from the software name. ADICS now refers to the protocol and infrastructure, whereas Attic is the concrete software implementation. Although the first incarnation of Attic (aka P2P-ADICS) was useful as a proof of the ADICS architecture, its limitations made integration with pre-existing Service and Desktop Grid architectures difficult. To solve these integration issues, the software presented here was constructed. It uses the lessons learned from the first software development and the ADICS architecture to extend the system to utilize HTTP for message interaction and use REST principles for exposing metadata on the network.

5.1.1 Introduction

A primary aim of the Attic software package is to be integrated easily into existing software stacks, both at a wire protocol level and at an API level. Hence the design of the system makes use of existing standards where possible. In particular, Attic uses HTTP exclusively, both for control messages and for data transfer. Unlike systems based on SOAP, which use HTTP to simply tunnel messages between participating parties, Attic takes a RESTful approach, making full use of what HTTP has to offer. There are two primary reasons for taking this design decision:

- Attic is about sharing binary data and HTTP is particularly well-suited to exchanging arbitrary data because it models all messages as a byte stream associated with a particular MIME type.
- Using HTTP allows the system to integrate easily with Web environments, in particular the Web browser. The Web browser is *the* primary interface to distributed systems for non-computer specialists. Providing access to the Attic system via this ubiquitously understood interface is of primary importance to any project involved in encouraging volunteer behavior among non-specialists.

Second, Attic allows its messages to be serialized to several widely supported data formats, specifically XML, Javascript Object Notation (JSON) [114], as well as XHTML. Supporting these different serialization mechanisms allows Attic data types to be integrated seamlessly into a number of environments ranging from Web browsers to Grid systems based on XML standards. Furthermore, at an API level, these technologies are supported across many programming and scripting languages. This makes integration with existing software stacks easier. Already, there are Attic components written for Java, C and the Unix shell.

An example of the benefits of this approach will be shown in the integration of the EDGI bridge component with the Attic network, described in detail in Section 5.2. This integration is extremely lightweight, requiring only components that are known already to exist on a bridge component.

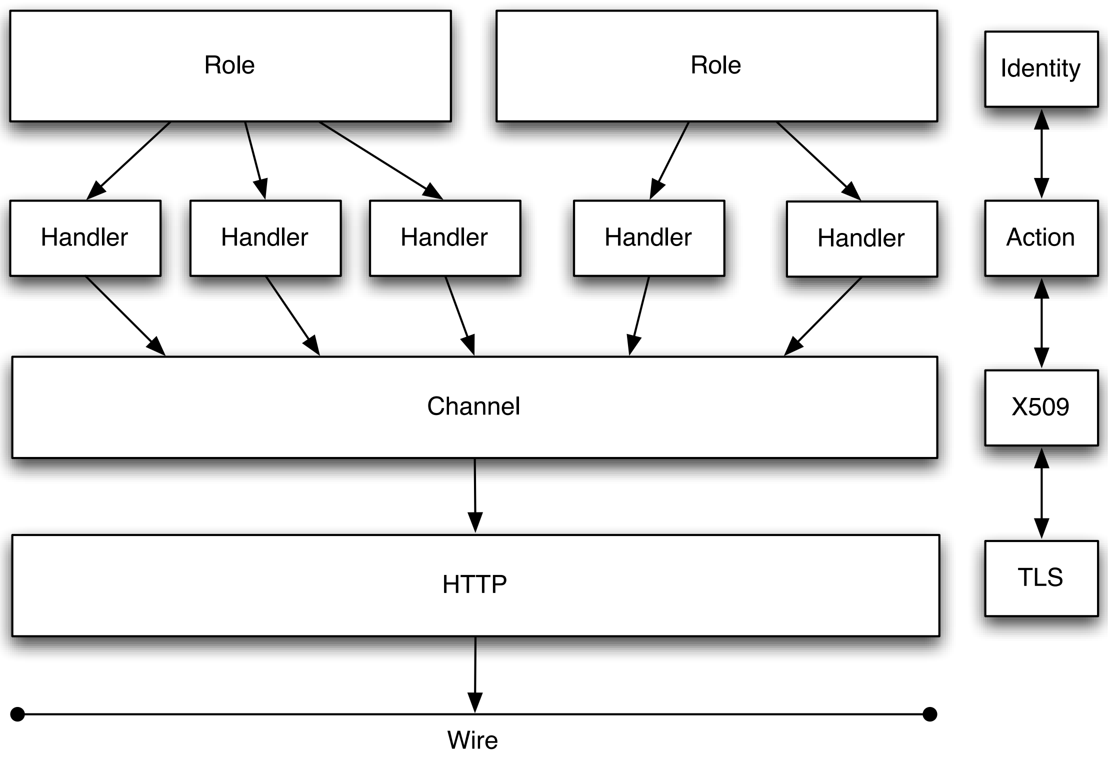


Figure 5.1: Attic Component Layers.

Figure 5.1 shows how the main components within Attic are layered. At the top of the stack is the concept of a *Role*. A *Role* encapsulates behavior that allows it to act out particular message exchanges in the Attic network. The primary *Roles* defined by Attic (based strongly upon the design given in Chapter 4) and referenced in this section are the following:

Worker — A node that downloads data and processes it on behalf of a Desktop Grid/volunteer computing project.

DataCenter (DC) — A node that provides caching facilities on the network and stores data.

DataProducer (DP) — A node that exposes data to the network.

DataLookup Service (DLS) — A node that allows producers and caching agents to register the data they have, and workers and caching agents to discover that data. A *DataLookup* may also take into account constraints provided at publish time when responding to discovery queries.

DataSeed — A node that allows brokered publishing of data. A *DataSeed* accepts data from a *DataProducer* and publishes it to a *DataLookup Service* on its behalf. In the process, the seed becomes the initial entry point for accessing the data.

A *Handler* is a component that responds to particular message events, for example data queries, or requests to download. Typically, a *Role* draws on the functionality of several *Handlers* to support its behavior.

The *Channel* layer provides an abstraction to the underlying message exchanges that take place over the wire. *Channel* components interface to the *HTTP* library. This library performs the actual data transfer.

Security runs vertically through the system. At the *Role* level, nodes are understood as *Identities*. An identity encapsulates a unique name, a token used to verify the identity, and a set of actions that are associated with the role. This allows an identity to be both authenticated and authorized to perform a particular activity. Currently, identities based on X.509 certificate chains are implemented. In this case, the unique name of an identity is the Distinguished Name (DN) as defined by the peer's certificate, and the token is the signed certificate chain including the peer's certificate and any authenticating certificates attached to it.

The action that is taking place during a particular request is defined at the *Handler* layer. This is determined by the URL that the request is directed to and the HTTP method being used. At the *Channel* layer, identities are understood as X.509 certificates. Each secured request contains a local certificate and a remote certificate. At the *HTTP* layer, these certificates are used to perform mutual authentication with Transport Layer Security (TLS) [115].

An overview of the roles and how they exchange messages is depicted in Figure 5.2.

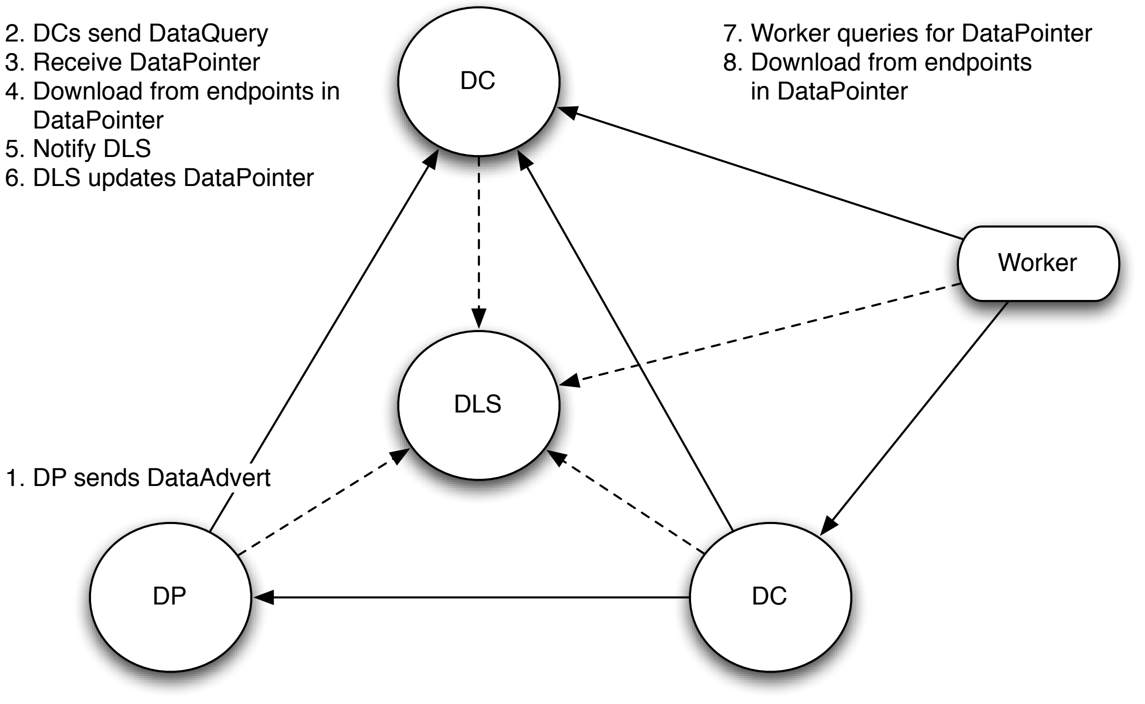

2. DCs send DataQuery
3. Receive DataPointer
4. Download from endpoints in DataPointer
5. Notify DLS
6. DLS updates DataPointer
DC
7. Worker queries for DataPointer
8. Download from endpoints in DataPointer
Worker
DLS
1. DP sends DataAdvert
DP
DC

Figure 5.2: Attic Protocol Workflow.

1. Data is published to a *DataLookup Service* using a `DataAdvert`. This may contain constraints such as the amount of replicas that are desired across the network and the lifetime of the data.

2. *DataCenter* nodes query the DLS for data to cache, using `DataQuery` messages. These may also be constrained; for example, they may specify maximum data size or project name.

3. In response to a query, a *DataCenter* receives a `DataPointer` containing endpoints associated with a metadata description.

4. The *DataCenters* proceed to download from the endpoints specified in the pointer.

5. Having downloaded the data, a *DataCenter* then notifies the DLS that is has the data, again using a `DataAdvert` message.

6. The DLS updates the pointer with the *DataCenter's* endpoint, adding it to the known list of replicas.

7. A *Worker* node then invokes an HTTP GET on a `DataPointer` endpoint.

8. The *Worker* proceeds to download from the endpoints specified in the pointer, in a manner identical to how a DataCenter retrieves a file.

5.1.2 Message Types

There are a number of message types used within Attic to allow describing, publishing, and querying data. They are currently serialized to wire as Javascript Object Notation (JSON) by default. XML and XHTML are also supported. JSON messages are between $\frac{1}{3}$ and $\frac{1}{2}$ smaller than XML, which is a motivator for choosing JSON as the default implementation.

The different message types and their usage are described below.

DataDescription — This type defines metadata about some data, for example a name, description, or project associated with the data. It also contains an identifier. This must be a Universally Unique Identifier (UUID) [116] (aka Globally Unique Identifier, or GUID). This could change if Attic adopted a hierarchical file system; however, at the moment, a UUID is required. A DataDescription also contains a FileHash.

FileHash — This type defines metadata about some data at the byte level. Specifically, it contains the length of the data and an MD5 hash of the bytes. It may also contain a list of FileSegmentHash objects.

FileSegmentHash — This describes a portion of data including the start offset and end offset in bytes, as well as an MD5 hash of the portion. A segment is created by a minter (e.g., a publisher) of a DataDescription who decides how big segments should be. The use of FileSegmentHashes allows for double-hashing, which improves network security and data integrity checking.

DataAdvert — A DataAdvert is used to publish the existence of data and the host that has that data. It is used by both publishing agents and caching agents. A caching agent is one that has previously retrieved information about data (in the form of a DataPointer), has downloaded the data, and is notifying the network that it now has the data. This type contains a DataDescription, an endpoint, as well as optional Constraints. A minter of a DataDescription should include a FileHash including segment hashes. When a caching agent notifies the network that it has cached the data, these details are not needed, because agents can match against UUID contained in the DataDescription.

Constraints — These are simple properties that can be added to a DataAdvert. They have a Type, one of String, Date, Integer, Double, Long or Boolean, as well as a string key and string value

DataQuery — This type is used to query for data. Like the DataAdvert, this can contain Constraints to restrict the types of data that are received from the query. The response to a DataQuery is a PointerCollection. A pointer collection contains one or more DataPointers. A DataAdvert may contain information that is not necessarily intended for public consumption, in particular the constraints. This is why adverts are not returned when agents query for data, but DataPointers are returned instead. Hence, logically, a DataPointer is created or amended when an advert is received by an agent that also responds to DataQueries, i.e., one that issues DataPointers on request.

DataPointer — This type contains a DataDescription and list of Endpoints. These are the endpoints from which the data can be retrieved. It is generated by services that have received any number of DataAdverts corresponding to a particular DataDescription. It is up to the client to decide which endpoints to use and which to forego.

PointerCollection — This type is simply a list of DataPointers.

Endpoint — This type is essentially a URI, but it can also contain another URI within it. When an endpoint appears in a DataPointer or DataAdvert, it may contain a meta URI. This is a URI from which metadata about the data to which the main URI points can be retrieved. This allows clients to query the particular endpoint to find out what segments it contains, in a manner very similar to BitTorrent. Currently, two query strings are supported, which, when appended to the meta URI return slightly different data.

The first is a *filehash* query, which returns the FileHash of a description referenced by the query value.

Filehash query to the meta endpoint *http://www.example.org/attic/meta*

```
http://www.example.org/attic/meta?filehash={description-id}
```

The second is a *description* query, which returns a representation of a full DataDescription.

DataDescription query to the meta endpoint
http://www.example.org/attic/meta

```
http://www.example.org/attic/meta?description={description-id}
```

The current implementation also supports path variants of these queries, for example:

```
http://www.example.org/attic/meta/description/{description-id}
```

and

```
http://www.example.org/attic/meta/filehash/{description-id}
```

5.1.3 Component Implementation

Attic is not tied to a particular HTTP implementation. The current implementation uses HttPeer [117], a HTTP library developed as part of the OMII funded WHIP project [118, 119]. HttPeer is a very lightweight library that allows easy server- and client-side HTTP data transfers. The Restlet framework [120] would also make a suitable HTTP back-end, although it is a larger code-base and some of the optimizations and security features could be harder to implement (being a third party library).

Attic is insulated from the HTTP implementation via a set of interfaces for sending and receiving data and control messages. These are based on the concept of `InChannels` and `OutChannels`. The former receive messages from the wire and pass them to server-side components. The latter are used by client-side components to push data onto the wire. Figure 5.3 shows the classes involved in the channel sub-system.

A `ChannelFactory` is used to create in and out channels. An `InChannel` takes a `ChannelRequestHandler` as an argument during creation. Request handlers receive messages along with metadata about the message when something arrives at the `InChannel`. Specifically, the handler receives a `ChannelData` object (not shown in Figure 5.3 for brevity). A `ChannelData` object is a container class

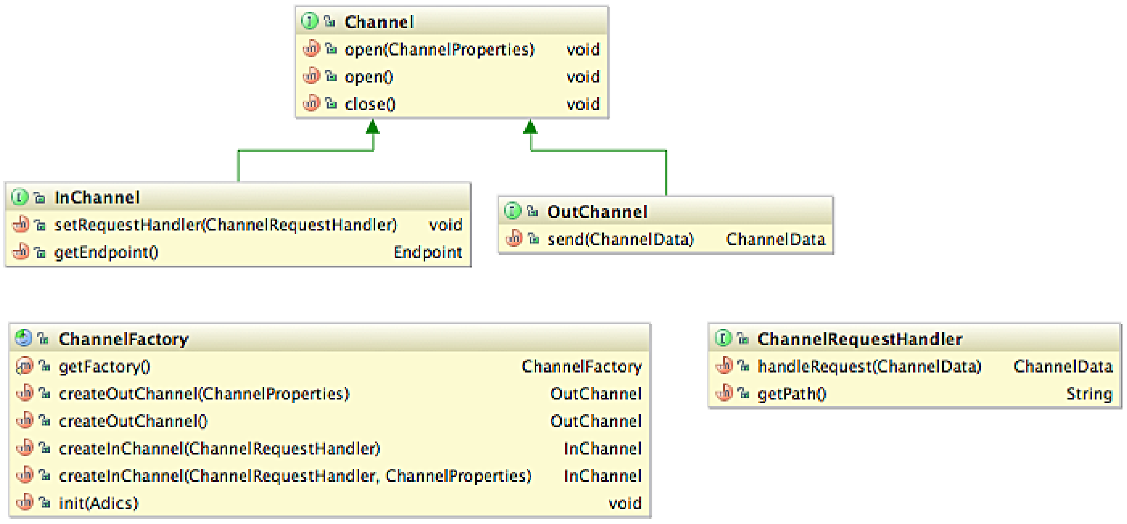

Channel
open(ChannelProperties) void
open() void
close() void
InChannel
setRequestHandler(ChannelRequestHandler) void
getEndpoint() Endpoint
OutChannel
send(ChannelData) ChannelData
ChannelFactory
getFactory() ChannelFactory
createOutChannel(ChannelProperties) OutChannel
createOutChannel() OutChannel
createInChannel(ChannelRequestHandler) InChannel
createInChannel(ChannelRequestHandler, ChannelProperties) InChannel
init(Adics) void
ChannelRequestHandler
handleRequest(ChannelData) ChannelData
getPath() String

Figure 5.3: Channel Interfaces.

with a number of properties such as the request endpoint, the request data object, the response data object, an action, — one of the HTTP methods — and an outcome — mapped to the HTTP response status code. `ChannelData` is used at both the server-side, by a `ChannelRequestHandler`, as well as at the client-side. On the client-side, the application asks a factory to create an `OutChannel` using the desired endpoint. After this is done, it inserts some data into it, and calls the channel's `send(ChannelData)` method. This returns synchronously, containing (possibly the same) `ChannelData` object with outcome and response data in it.

The system uses a typical HTTP architecture in which different handlers are mapped to different URL paths (note the `getPath()` method of the `ChannelRequestHandler` interface). Apart from this, a handler handles a request using the `handleRequest(ChannelData)` method.

Extending the `ChannelRequestHandler` is the `AbstractRequestHandler` class. This provides some functionality for handler implementations, in particular breaking the request into HTTP method actions. Hence subclasses of this abstract class do not have to explicitly work out the request type. Instead, they can process messages directly in a Servlet framework, much in the way that an extension of abstract `HttpServlet` can.

The existing extensions to the `AbstractRequestHandler` are shown in Figure 5.4. Their roles are enumerated below:

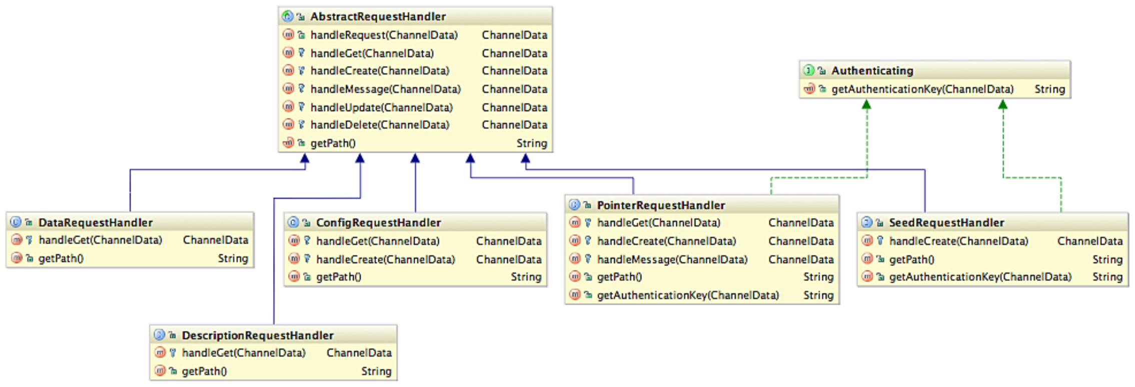

AbstractRequestHandler
handleRequest(ChannelData) ChannelData
handleGet(ChannelData) ChannelData
handleCreate(ChannelData) ChannelData
handleMessage(ChannelData) ChannelData
handleUpdate(ChannelData) ChannelData
handleDelete(ChannelData) ChannelData
getPath() String
Authenticating
getAuthenticationKey(ChannelData) String
DataRequestHandler
handleGet(ChannelData) ChannelData
getPath() String
ConfigRequestHandler
handleGet(ChannelData) ChannelData
handleCreate(ChannelData) ChannelData
getPath() String
PointerRequestHandler
handleGet(ChannelData) ChannelData
handleCreate(ChannelData) ChannelData
handleMessage(ChannelData) ChannelData
getPath() String
getAuthenticationKey(ChannelData) String
SeedRequestHandler
handleCreate(ChannelData) ChannelData
getPath() String
getAuthenticationKey(ChannelData) String
DescriptionRequestHandler
handleGet(ChannelData) ChannelData
getPath() String

Figure 5.4: Channel Handlers.

ConfigRequestHandler handles requests for managing the Web configuration. It responds to the GET method by returning the Web form, and likewise processes the form's POST requests.

DescriptionRequestHandler handles requests for metadata, specifically DataDescriptions and FileHashes

DataRequestHandler handles requests for data. It deals only with GET methods at the moment. There is the possibility that it could handle queries in the form of an HTTP POST. This would allow clients to specify bandwidth constraints on data delivery by the server.

PointerReqestHandler handles requests dealing with DataPointers, DataAdverts, and DataQueries. There is a fair amount of decision making to be done with these types, so this handler delegates most of the logic to an AdvertProcessor (see below).

SeedRequestHandler handles requests to publish data. Typically, a data publisher will act as a seed itself, but this is not always the case. A publisher may push data to a node prepared to receive both a DataAdvert and the associated data. The seed then becomes the publisher itself, passing the DataAdvert to the network and storing the data locally.

Some handlers implement the `Authenticating` interface. This interface is used to specify a request action that is being performed by the client, based on the request path and the HTTP method being used by the client. These actions are mapped to identities, specifically the Distinguished Name of the client's X.509

certificate. The security implementation is discussed in more detail in §5.1.7. Actions include core functionality such as PUBLISH and CACHE.

The other core interface along with ChannelRequestHandler is the Role interface. A Role has a reference to an instance of an Attic object. This is the central point of entry for an application, providing the glue between configuration options and different roles. A Role is initialized with an instance of Attic. It does not expose any service capabilities, although strictly speaking, it could. The basic implementation of a *Worker* node implements the Role interface. The worker does not require any running servers. It merely makes client requests.

Extending the Role interface is the ServiceRole interface. This is designed for nodes that provide a service to the network. As well as extending the Role, it also extends ChannelRequestHandler. Therefore it binds an instance of Attic with the server-side message receiving interfaces. A ServiceRole is designed to create a channel and register as a handler with it, but to delegate the processing to other handlers based on the request path and the path of its registered handlers. ServiceRole exposes an addChannelRequestHandler method, which allows a service to add arbitrary handlers to its own path to provide capabilities.

The AbstractServiceRole provides some core methods for services to extend. In particular, it knows how to determine which registered ChannelRequestHandler should be invoked, based on the request path of the ChannelData, as well as checking the authentication key of the handler. If it returns one based on the request path, it is compatible with the identity that is making the request. This allows subclasses to know that, if the request gets as far as them, then the requesting agent has been authorized and authenticated. To support this behavior, ServiceRole exposes addIndentity and removeIdentity methods.

Two further extensions to the ServiceRole interface are currently defined. The Publisher interface is implemented by roles that initiate data onto the network. A publisher typically creates an authoritative metadata description which is registered with the network and downloaded by caching and worker nodes. Registration happens via the publish(DataAdvert) method.

The AdvertProcessor interface is implemented by nodes that receive DataAdverts and DataQueries during their message processing. Typically, this is a role taken on by a DataLookup Service. The two methods receive an advert or query, and the ChannelData is used to process the request. It is the responsibility of the AdvertProcessor to manipulate the ChannelData and set the response data and outcome of it.

5.1.4 Role Implementations

The way roles are defined in the interfaces and utility implementations of `ServiceRole` and `ChannelRequestHandler` means that there is much flexibility in how the different capabilities are combined to create nodes that perform particular services to the network. This is to allow for flexible network conditions. For example, in a centralized system, there may be a single authorized node that performs the registration and exposure of published data. In a decentralized network, on the other hand, nodes may publish their data to their neighbors or to a federation of look up services. These differences imply that the logic implemented by nodes may vary, in terms of security and the message exchanges that nodes need to support.

Currently supported roles are:

DataWorker — This component knows how to pull data given a remote endpoint that references a `DataPointer` document. It implements the `DataReceiver` interface which is notified when data has finished downloading. It does not support any persistent services other than the configuration service described next.

ConfigServiceRole — This is a Web configuration service that can be attached to an `Attic` instance to provide online configuration. It runs on the default port of an initialized service (or falls back to 28842) unless the port number has been set in the configuration itself. It is available at the absolute path `/attic/config` on the host on which it is running.

DataPublisher — The `DataPublisher` component can index local files and publish `DataAdverts`, given a bootstrap endpoint to which to publish to. It implements the `Publisher` interface in addition to extending `AbstractServiceRole`. Indexing involves taking a File or Directory and a template DataDescription specifying metrics such as project and file size, and creating a full metadata description with data chunk hashes. During the process of indexing, any files being indexed are renamed to the UUID in the resulting DataDescription. By default, the DataDescription is written to disk, allowing the mapping between a file and a description to be discovered again on start up. The default location to put files to be indexed is `<attic-home-directory>/<role>/data`, where role is the role of the component storing data. For a DataPublisher this is *dp*, for a DataCenter, this *dc*, and for a Worker this is *dw*. The default location for writing out matching DataDescriptions is `<attic-home-directory>/<role>/desc`. For example, if the Attic home directory is `~/.attic`, which is the default on Unix-like machines, then the data directory for a Publisher would be `/.attic/dp/data` and the directory in which DataDescriptions are stored would be `/.attic/dp/desc`

DataSeed — The DataSeed component extends the DataPublisher by adding a SeedRequestHandler to its list of message handlers. Subscribing to the role of a seed handler allows remote clients to push data to it. This data is then published by the seed to the network. The handler first accepts a DataDescription which is expected to be a template, that is, excluding a list of segments. Any identifier, if present, will also be overwritten and replaced by a generated UUID. In response to receipt of a description, the seed handler returns an endpoint on the seed node of where to send the actual data. The client then uses an HTTP POST to send the data. When the seed has received the data, it publishes the data description to its look-up service (i.e., DLS) as a DataAdvert. Contained within this advert is the location of the seed as an endpoint. In response to its publication, the seed gets an endpoint to a DataPointer from the look-up service. The seed then forwards this to the client.

Currently the seed returns an `attic` or `attics` URL to the client. These URL schemes are used to allow transparent data access from an endpoint referencing a DataPointer document. The `attic` scheme is used for non-secured endpoints (HTTP) and the `attics` scheme is used for secured (HTTPS) endpoints. Changing the scheme of the URL to its HTTP variant will allow normal retrieval of the data pointer from the remote endpoint.

DataLookup — This component acts as a look-up service (aka, the Data Lookup Service) for other nodes. It implements the AdvertProcessor interface as well as extending AbstractServiceRole. In particular, it accepts requests to publish DataAdverts and requests to cache data via DataQueries. The current implementation provides the path `/dl/meta/pointer`, where requests to publish and cache can be directed via a POST request, e.g., `https://example.org/dl/meta/pointer`. This is the publish URL when a caching agent has downloaded data. Following the download, the new replica notifies the look-up service with a DataAdvert in a POST request. The endpoint the request is directed towards is the same endpoint of the existing pointer. For example, `https://example.org/dl/meta/pointer/1234567890`[2] is a DataPointer URL, and when a DataLookup service receives a publish request containing a DataAdvert, the implementation does the following:

- If the request is directed at the publish URL and no matching DataAdvert is found in the local cache, the DataAdvert is cached and a new DataPointer is created from the description and the endpoint in the advert. If a matching DataAdvert is found locally, an associated local DataPointer will also exist. If this is the case, then this is a request to update or re-publish the DataPointer. The DataDescription in the DataAdvert is copied to the local DataPointer, as well as the endpoint in the advert. Existing endpoints in the pointer are not removed. The assumption is made that the data being referenced by the advert and pointer has not changed, otherwise a new DataAdvert should be published.

  Updating or republishing an advert is subject to ownership constraints. Ownership of a DataAdvert is currently expressed via the identity defined on a secure connection. That is, the identity of the requester is determined by the certificate being used in the communication and is mapped to the advert at publish time. If an update is requested to a DataPointer that has an identity mapping and the update request does not contain an identity (i.e., is not performed over a secure connection), or the supplied identity does not match, the update will fail. If the identities match, then the request has come from the minter of the DataAdvert and therefore authority exists to manipulate the metadata. Note that deleting an advert, and hence the DataPointer associated with it, can also be done only if the initial advert was published using an identity and the request to delete it has a matching identity.

  These constraints are overridden if the Attic security configuration `setTestMode` property is set to “true” (see §5.1.8 for details).

- If the request is directed at the pointer URL, a mapped DataPointer

[2]Note that under the current implementation, a UUID for an Attic URL would not likely result in `1234567890`, but rather a hash-like string such as `09aa632b-0031-42be-a2e2-49c706e704d8`. For space considerations, `1234567890` is used to denote an Attic identifier in this text.

should already be cached by the look-up service. If not, an error is returned. Otherwise, the endpoint in the DataAdvert is added to the endpoints in the extant DataPointer.

To remove itself from the list of endpoints in a particular pointer, an agent can send a DataAdvert with a constraint key of *dereference* and a value of "true."

- When the implementation encounters a DataQuery, it expects the query to be directed at the publish URL, not a pointer URL. This is because a request to cache is for any data, not a particular file. The term any is qualified by the Constraints set on the query itself. The response to a DataQuery is a PointerCollection. This is a list of DataPointers that match the query. Note that matching and constraints are currently implemented in base forms, and would need to be expanded to support enhanced scenarios. The main constraint that is currently implemented is the replica constraint on a advert. This value restricts the number of times a data pointer is issued in response to a query. Other than that a very simple equality match is currently made between advert constraints and query constraints. For example, if both share a project constraint with a string type this is equal, then the query will match the advert. Constraints have types (see §5.1.2) which can be used to express slightly better matching semantics, such as before and after for date types, and less than or greater than for number types.

DataCenter — A `DataCenter` acts as a client in that it requests DataPointers from a `DataLookup` service and downloads the data from the endpoints described in the pointers. It should be noted that a DataCenter does not typically process any data, but simply caches it. Once it has downloaded the data, it sends a DataAdvert to the DataLookup Service, in the hope that the look-up service will add its endpoint to the relevant DataPointer. DataCenters are also the nodes that send DataQueries to the look-up service; hence they receive a list of DataPointers in response to the query.

The `Role` interface exposes methods that attach it to an instance of an `Attic`. The `init(Attic)` method is called to pass an instance of `Attic` to the role, providing it with a central point to retrieve configuration information. This allows the new entity to launch any components it needs to in order to fulfill its tasks. Similarly, the `shutdown` method is used to close down any components and clean up.

Note that while these methods can be called directly on `Role` implementations, this is not recommended because the `Attic` instance itself needs initializing at some point. Instead, client code should create an instance of `Attic` and call the `attach(String, Role)` method on it, passing in the role, and a string identifying it. Then, by calling the `init` method on the `Attic` instance, the role's `init` method will be called within that, at the correct time of initialization. The applies to the `shutdown` method of `Attic`.

5.1.5 Downloading

Because the richness of new features that could be added is almost unlimited, the aim of the downloading sub-system design is to be as extendable as possible considering the various dependencies and relationships between components. This enables Attic to be enhanced for the particular use-cases of each project that wishes to employ it, a key design feature.

The process of downloading data begins with a `DataPointer`, as downloading from multiple servers using Attic requires that one is in possession of a `DataPointer`. Typically, these can be retrieved from the DataLookup Server, although they could theoretically be distributed out-of-bounds. DataCenters often query for a collection of available pointers because they are interested in caching, as opposed to processing the actual data. Workers, on the other hand, will query for a pointer based on the UUID (or GUID) of the `DataDescription` that is referenced by the pointer, because they require a particular data object to process. As described in §5.1.2, a `DataPointer` represents a description of data and a list of `Endpoints` that potentially have some or all of the referenced data. The `Endpoint` element represents the URL at which the data is available. It can optionally contain a meta URL. This is a URL from which metadata about the data (either a `DataDescription` or a `FileHash`) can be retrieved. An example XML serialization of a portion of a `DataPointer` is shown in Listing 5.1. For comparison, its JSON equivalent is given in Listing 5.2.

Listing 5.1: XML Rendering of a DataAdvert

```xml
<?xml version="1.0" encoding="UTF-8"?>
<DataAdvert xmlns="http://atticfs.org">
   <DataDescription xmlns="http://p2p-adics.org">
      <id>12c667d6-2d5d-4904-9c2c-6746251b81ef</id>
      <name>SimulationInputFile.dat</name>
      <project>EDGeS</project>
      <description>Input for simulation</description>
      <FileHash>
            <hash>661c7f5e462be8ced9a8a6d8a1c7e6</hash>
            <size>12407432</size>
            <Segment>
               <hash>bedbfd11fa5fb4fd6b97349f45b6b3</hash>
               <start>0</start>
               <end>524287</end>
            </Segment>
            ...
            ...
            ...
            <Segment>
               <hash>32ce5368d98243a2a9abeccc2ddc5c</hash>
               <start>12058624</start>
               <end>12407431</end>
            </Segment>
      </FileHash>
   </DataDescription>
   <Constraints>
        <Constraint type="Date">
            <key>expires</key>
            <value>Sat Jun 30 23:59:59 GMT 2012</value>
         </Constraint>
         <Constraint type="Integer">
            <key>replica</key>
            <value>3</value>
         </Constraint>
      </Constraints>
</DataAdvert>
```

Listing 5.2: JSON Rendering of a DataAdvert

```
{
"DataAdvert": {
   "DataDescription": {
       "id": "b426c41b-d5a3-4138-93ec-60a8be2a6c0c",
       "name": "SimulationInputFile.dat",
       "project": "EDGeS",
       "description": "Input for simulation",
       "FileHash": {
           "hash": "661c7f5e462be8ced9a8a6d8a1c7e6",
           "size": 12407432,
           "Segment": [
               {
                "hash": "bedbfd11fa5fb4fd6b97349f45b6b3",
                "start": 0,
                "end": 524287
               },
               ...
               ...
               ...
               {
                "hash": "32ce5368d98243a2a9abeccc2ddc5c",
                "start": 12058624,
                "end": 12407431
               }
           ]
       }
   },
   "Constraints": {
       "Constraint": [
           {
               "type": "Date",
               "key": "expires",
               "value": "Sat Jun 30 23:59:59 GMT 2012"
           },
           {
               "type": "Integer",
               "key": "replica",
               "value": "3"
           }
       ]
   }
}
}
```

The first step in the download process is to contact all the endpoints that expose a meta URL along with the data URL. A query is sent to the meta URL for a list of the hashed segments stored by each host. This series of queries is performed by a `RequestResolver`. The output of the process results in a `RequestCollection`. This is a list of `EndpointRequest` objects. An `EndpointRequest` is a mapping of many hashed segments to a single host. A `RequestCollection` orders the requests according to the Round Trip Time (RTT) of the request for hashed segments. This allows the client to ascertain how fast each server is currently responding. It also allows the Worker to weed out any endpoints that are not responding at all, or return an error status to the request, for example, an HTTP Not Found status code. Endpoints that did not supply a meta endpoint are added as reserve mappings to the collection and used as a last resort during downloading. Hence, not supplying a meta endpoint results in the host's always being far down the list of potential endpoints. This mechanism is used by the DataSeed component to de-prioritize itself.

It should be noted that the RTT-based selection of DataCenters can be enhanced with more complex and intelligent scenarios that can optimize download times. For example, a promising Quality of Service (QoS) enhancement for Attic that uses peer-reporting mechanisms to keep track of download heuristics was implemented as a Ph.D. project at Cardiff University (see Appendix B).

Downloading data based on a `RequestCollection` is done using a `Downloader` instance. A `Downloader` takes, or creates a `DownloadTableCreator` instance. A `DownloadTableCreator` takes a `RequestCollection` and creates a `DownloadTable` from it. This table contains a queue of `SegmentRequest` objects. These are mappings of a single endpoint to a single hashed segment as defined in the `DataDescription` element of the original `DataPointer`. They also contain a list of sub-segments. These sub-segments are the actual chunks that are downloaded and may be smaller than the hashed segment defined in the data description. `SegmentRequests` are prioritized using the policy of the table creator and a `SegmentRequestComparator` implementation that is supported by the `DownloadTable` instance being created. The two currently supported policies are to either split the chunks evenly between available endpoints and prioritize according to the speed of the RTT metrics, or to split the chunks between the available endpoints based on the index of the chunk in the file. This latter policy is used by the `AtticInputStream` for the Attic URL implementation (see §5.1.6). It ensures that data arrives roughly in the order in which it is defined in the metadata, meaning less reordering needs to happen within the stream exposed to the client application. This is a critical step for providing the InputStream needed by clients such as XtremWeb (see §5.4).

Next, a template view of the data is created. It is possible that not all the segments of a file are currently available and therefore it cannot be downloaded to completion. This template is matched against by the `DownloadTable` during the download process to determine the point at which all possible data has been retrieved.

Once a `DownloadTable` instance is available, the downloader uses implementations of the `AbstractRequestor` to remove `SegmentRequests` from the table's queue. The amount of concurrent threads performing download is determined by the `getMaxFileConnections` configuration parameter of the Attic download config (see §5.1.8). When a requestor has completed the download of a sub-segment contained in a `SegmentRequest`, it notifies the table of either success or failure. Depending on policy, a retry may be attempted, or an alternative endpoint with the same hashed segment is located in the table and used to complete the download. This process continues until the status of the data being rebuilt matches the status of the data as defined during construction of the `RequestCollection`. If a full set of hashed segments was attainable from the endpoints provided in the `DataPointer` then the `COMPLETE` status should be achieved during download. If a table cannot achieve the initial status, and the table is exhausted, with no more possible endpoints to download from, then the table completes anyway, leaving the downloaded data in an unfinished state. Such an occurrence would result in the return of an appropriate status such as `DISCONTINUOUS`, signaling that there are gaps in the downloaded data, or `CONTINUOUS`, meaning data is either missing from the beginning of the data or from the end of it.

Figure 5.5 shows the process of converting a `DataPointer` into a prioritized queue of requests for individual chunks. The first two endpoints in the pointer contain meta URLs, so these are queried for the chunks they have and ordered according to their RTT. The second endpoint has a faster RTT, so it appears earlier in the list of `EndpointRequests`. The third endpoint has no meta URL, so it is added to the end of the list, logically containing all segments; therefore, an endpoint with no meta URL is presumed to have all segments. The `EndpointRequests` are then converted to `SegmentRequests`. The policy for ordering segment requests shares out segments as evenly as it can between servers. Hence *Segment 2* that is available at the slower endpoint is prioritized above the *Segment 2* at a faster

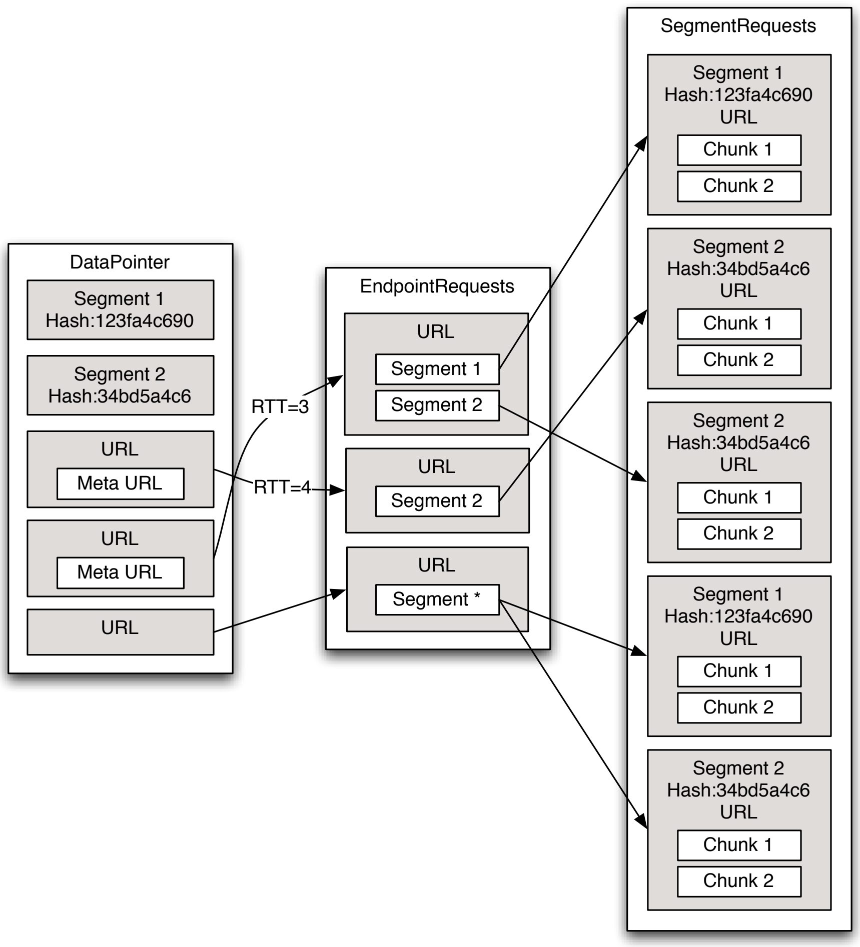


Figure 5.5: Segmenting Data.

endpoint. This allows the client to concurrently download both segments from different servers. The segments that are possibly available at the endpoint with no meta URL are added as back-up requests to be attempted as a last resort.

From an application perspective, downloading is done by passing a `RequestCollection` and an instance of the `DataReceiver` interface to a downloader, then calling its download() method. The `DataReceiver` receives notification when download has completed.

5.1.6 Attic URLs

To simplify integration with other systems (e.g., XtremWeb), the Attic implementation supports the concept of an Attic URL. These are URLs that use an Attic-specific scheme and can therefore be attached (e.g., in Java) to an Attic protocol handler. Two types of scheme are supported — `attic` for unsecured transfer and `attics` for secured communications. Along with this URL scheme, the implementation provides an extension to the `URLStreamHandler` class defined by Java. A stream handler is called upon by a `java.net.URL` instance to create a connection to the location defined by the URL string. Using the Attic protocol/stream handler allows applications to create a URL with an `attic` scheme and read directly from the URL's input stream, with only very minor modifications to their code (i.e., registering the protocol handler and including the Attic jar file).

An Attic URL ultimately points to a `DataPointer` at an HTTP endpoint. This means that by substituting the `attic` scheme with `http`, and dereferencing it, one can directly receive the requested document. For example, the standard attic URL for any file will result in its `DataPointer` document. Therefore, what an `attic` URL ultimately provides is a pointer to comprehensive metadata that references multiple endpoints; those endpoints themselves reference the actual data that can be obtained using http/https.

When an application calls the `URL.openStream()` method on an Attic URL, a new input stream is created. Underneath, it contains multiple input streams corresponding to the endpoints defined in the `DataPointer`, which are concurrently downloaded, reassembled into the correct order, and passed back through the requesting input stream. The same downloading techniques are used as when downloading to the local file system, with the exception that the streams are returned directly to the application rather than being written out to file.

The fact that the streams are passed directly back to the application limits the possibility for verifying the data against the MD5 hashes defined in the metadata. The policy taken by the stream (which can be configured) is to attempt verification if the hashed chunks defined in the metadata are less than or equal in size to the set in-memory buffer. If the hashed chunks are too big to verify in memory, then the application must take on this responsibility.

5.1.7 Security

Security in Attic is currently implemented using TLS and mutual authentication with X.509 certificates. To enable the security features of Attic, Java keystores must be available at runtime and contain local keys and certificates, as well as trusted certificates. These can be set using the security configuration options.

While signed certificates provide authentication of an entity, they do not allow for more fine-grained authorization of actions based on the identity in the certificate. To achieve this, Attic uses a mapping between an application-level defined action and a certificate's Distinguished Name (DN).

The `ServiceRole` interface, exposes an `addIdentity` and `removeIdentity` method. The `Identity` interface encapsulates a unique identity, an object that is used to determine the authenticity of the identity, along with a list of supported or allowed roles. Two implementations currently exist — `X509Identity` and `DNIdentity`. The first of these is used when a message arrives at a service. The `ChannelData` is populated with both the remote and local certificates (if any) being used during the transaction. These can be retrieved using the `ChannelData` methods `getRemoteIdentity` and `getLocalIdentity` respectively. The `DNIdentity` is currently used by the default in-memory identity implementation that stores Identities and their allowed roles. Equality between the distinguished name of the two types of `Identity` is used to determine whether the currently connected peer is allowed to perform a particular action. The current *action* is determined by querying the appropriate `ChannelRequestHandler`, provided it implements the `Authenticating` interface. Hence a correlation between the `Authenticating` instance's action and the roles allowed by a particular `Identity` determine whether or not a peer may perform a particular request.

5.1.8 Configuration

An instance of Attic comes with a set of configuration classes dealing with different aspects of the system. These are listed below.

`org.atticfs.config.security.SecurityConfig` deals with security-related configuration.

setSecure(boolean):void — whether to use secure connections or not (i.e., HTTP or HTTPS).

setRequireClientAuthentication(boolean):void — whether or not to require clients to authenticate themselves with a certificate if using secure connections.

addKeyStore(Keystore keystore):void — this adds a keystore (local private key and matching public certificate in a Java keystore) to the security config. When a secure socket is created, the local keystores are used to locate a certificate for the local peer. Attic uses the standard Java mechanism for storing keys and certificates in keystore files. The `org.atticfs.config.security.Keystore` class is a container for data relating to a Java keystore. It includes properties for specifying the location of the keystore, (e.g., file path, keystore password, key password, keystore alias, keystore type, and encryption algorithm). An `org.atticfs.config.security.Keystore` can also be mapped to an authority (host/port combination). This allows different keys to be used depending on the port the server is running on. A value of *default* means the keystore will be used for any connections that do not specify a host and port matching the current endpoint.

addTrustStore(Keystore keystore):void — this is similar to the `addKeystore` method, adding a trust keystore. These are used when creating a secure socket. The certificate of the remote peer is matched against certificates in the trust stores. A certificate will be allowed if it appears in the trust store or is signed by a certificate in the trust store. By default, Attic also trusts the default certificate authorities that ship with the current Java installation.

setTestMode(boolean testMode) — primarily designed for debugging, this allows the deletion and updating of pointer metadata without the need for an identity to be provided in the request. The default value is "false." It also supports the listing of all data pointers currently stored by a DataLookup Service, wrapped in a PointerCollection container.

`org.atticfs.config.data.DataConfig` deals with data-related configuration, for example, setting hashed chunk sizes and specifying how much local disk space should be provided to Attic for data storage.

setMaxLocalData(long maxLocalData) — this sets the total amount of space (in bytes) that Attic is allowed to use on downloaded data. This is relevant for DataCenter nodes in particular. The default value is 100 GB.

setFileSegmentHashSize(int fileSegmentHashSize) — this sets the size (in bytes) for hashed segments of data. The default value is 500 KB.

setWriteDescriptionsToDisk(boolean writeDescriptionsToDisk) — this determines whether to write out metadata descriptions to disk when indexing files. By default this is true. Writing descriptions out means they will not be indexed next time the node starts up.

setDataQueryInterval(long dataQueryInterval) — this sets the interval between querying for data in seconds. This is used particularly by DataCenter nodes. The default query interval is one hour.

`org.atticfs.config.download.DownloadConfig` deals with download-related configuration, for example, how many threads to allow per download.

setStreamToTargetFile(boolean streamToTargetFile) — if set to "true," when downloading, interim files will not be created. Instead data will be written directly to the final file. Downloaded blocks that do not pass verification are marked as invalidated portions of the final file until a segment download for that file succeeds.

setBufferSize(int bufferSize) — this sets the in-memory buffer size while downloading.

setMaxTotalConnections(int maxTotalConnections) — this sets the maximum total connections (i.e., threads) allowed by a client during downloading.

setMaxFileConnections(int maxFileConnections) — this sets the maximum total connections (i.e., threads) allowed by a client per file.

setConnectionIdleTime(int connectionIdleTime) — this sets the idle time on open connections. This is used at the server side to close connections that have been left open by clients. The default value is three minutes.

setDownloadChunkSize(int downloadChunkSize) — this sets the size of chunks to download. These are typically smaller than hashed chunks as defined in the metadata. This means multiple smaller chunks are downloaded before a single hashed chunk of data can be verified. The default size is 256 KB.

setRetryCount(int retryCount) — this sets the number of times a failed download for a chunk is retried (from the same server). The default value is two.

`org.atticfs.config.stream.StreamConfig` deals with streaming-related configuration, for example, whether to attempt verification of chunks before returning them to the application. This is used by the `atticConnection` class that extends URLConnection and is used for the `attic` URL scheme.

setAttemptVerification(boolean attemptVerification) — this determines whether to attempt verification of data before passing it to the application stream. Actual verification depends on the hashed chunk size defined in the DataDescription that is being downloaded and the size of the in-memory buffer described below.

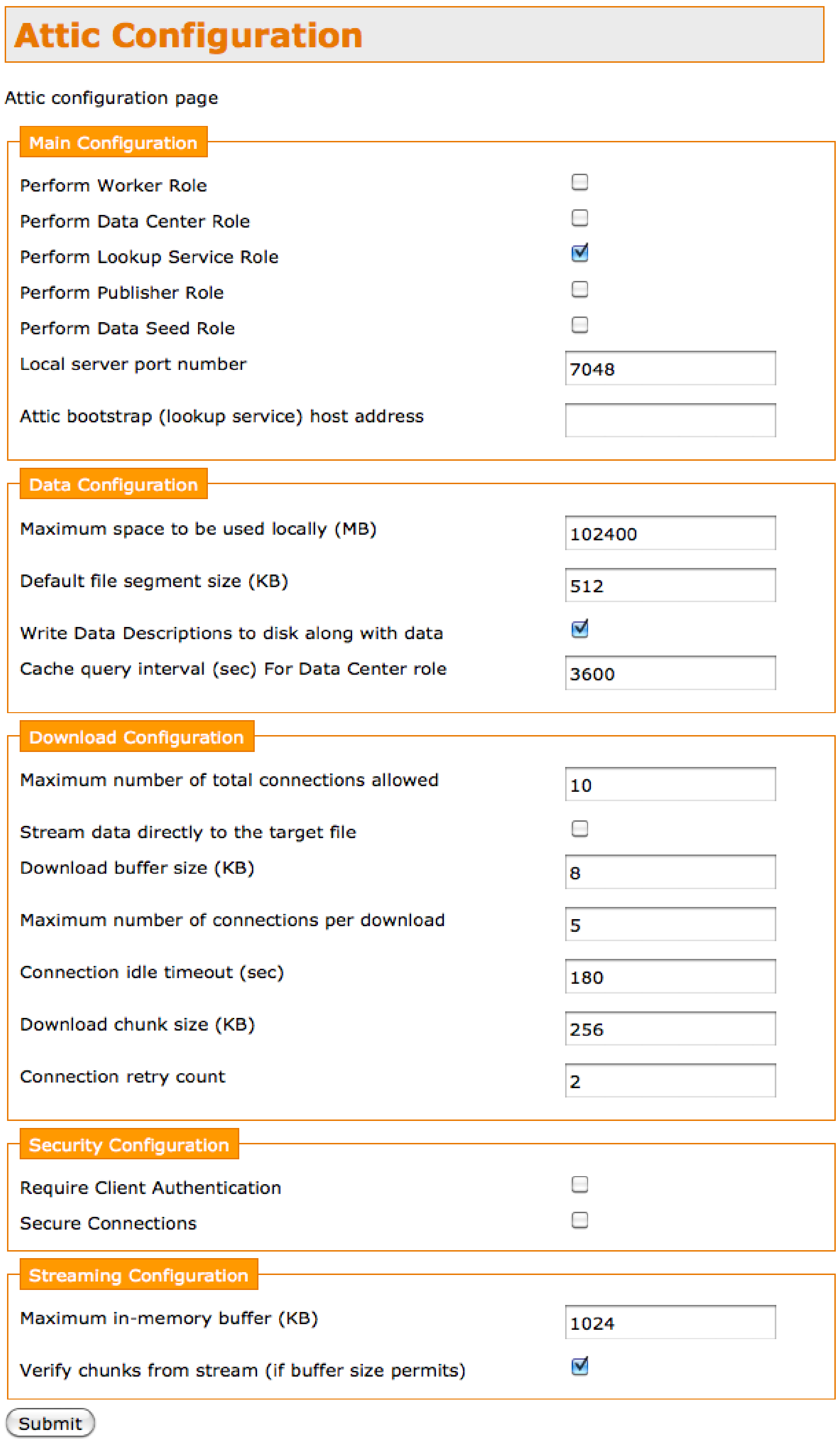
setMaxBufferSize(int maxBufferSize) — this sets the maximum in-memory buffering of streaming data. If this value is greater than or equal to the size of hashed chunks in the data, and attempt verification is set to "'true," data will be verified before passing it to the application stream. Verification failures are retried.

When using Attic simply as a URL protocol handler, these properties can be directly set in the `atticConnection` class using the URLConnection method `setRequestProperty(String, key, String value)`. The memory buffer key is `org.atticfs.protocol.attic.max.inmemory.buffer` and the verification attempt key is `org.atticfs.protocol.attic.attempt.verification`. The values can be given as strings, e.g., for the int 10, "10" and for the boolean true, "true."

`org.atticfs.config.html.HtmlConfig`. This class wraps the above-specified configuration objects and exposes them via an HTML interface, as shown in Figure 5.6. It is used by the ConfigServiceRole and ConfigRequestHandler to expose a Web interface to the configuration options. This interface is typically available at the path `/attic/config` on a running instance of Attic and allows easy system administrator configuration of Attic options.

5.1.9 Summary

The preceding section has given an overview of the Attic software. Attic is the second generation (and current) implementation of the ADICS research and design specified in Chapter 4. The Attic implementation has proven very useful for data integration with current Desktop Grid middleware, as will be shown in §5.2. Naturally, software and requirements evolve and there are enhancements that could



Attic Configuration

Attic configuration page

Main Configuration

Perform Worker Role	☐
Perform Data Center Role	☐
Perform Lookup Service Role	✓
Perform Publisher Role	☐
Perform Data Seed Role	☐
Local server port number	7048
Attic bootstrap (lookup service) host address	

Data Configuration

Maximum space to be used locally (MB)	102400
Default file segment size (KB)	512
Write Data Descriptions to disk along with data	✓
Cache query interval (sec) For Data Center role	3600

Download Configuration

Maximum number of total connections allowed	10
Stream data directly to the target file	☐
Download buffer size (KB)	8
Maximum number of connections per download	5
Connection idle timeout (sec)	180
Download chunk size (KB)	256
Connection retry count	2

Security Configuration

Require Client Authentication	☐
Secure Connections	☐

Streaming Configuration

Maximum in-memory buffer (KB)	1024
Verify chunks from stream (if buffer size permits)	✓

Figure 5.6: Configuration Interface.

be made to Attic that would increase both its usability and its functionality. Several of these features and enhancements have been identified and are discussed in the Future Work section of Chapter 6.

5.2 Data Migration Using Attic

The following section gives an overview of the data paths from the EDGI Service Grid infrastructures to the Attic network. Specifically, this section focuses on how data moves to Attic from a target SG through the bridge, and is then made accessible to client machines. Each of the three target Service Grids of EDGI, gLite (see §5.2.1), ARC (see §5.2.2), and UNICORE (see §5.2.3) are described in this section, along with their needed modifications to transition data to Desktop Grids. It should be noted that much of the integration work described here is the result of collaborative efforts in the EDGeS and EDGI projects. Design and architecture were often co-developed, with the Attic enhancements, Service Grid adapters, and 3G Bridge enhancements being performed by different partner institutions proportional to expertise and project involvement.[3]

It is useful to describe the enhancements and needed infrastructure changes here, as they show how the Attic software, and ADICS architecture, has been adapted to real-world situations. Overviews of each Service Grid data infrastructures are given to provide an idea of the heterogeneity of the target systems and to show how these diverse Service Grids were able to adapt to conform to a generic data transition mechanism. This discussion is especially useful as it shows how the design decision to use a new protocol and REST principles eases integration and provides flexibility.

5.2.1 Data Transition from gLite $\rightarrow$ DG

When a job is moved from gLite to a Desktop Grid for execution, there is a need for the job to receive its input data. gLite input data is generally stored in user-accessible and secured locations accessible through the gLite Storage Resource Management System (SRM). Users have Logical File Names (LFNs) that relate

[3]Please note that several of the images, as well as textual descriptions of SG$\rightarrow$DG job migration in this section, were taken from EDGI's D6.1 deliverable entitled "D6.1 Supplementary report on: Data distribution paths from SGs to DGs," which I coordinated. [121]

to their files and they are able to query the SRM for concrete locations of their files. This process generally requires authentication, although the files could also be found in other locations outside of the SRM. For data distribution to Desktop Grids, certain input files can be passed directly through the 3G Bridge [122][4] (e.g., smaller or infrequently used files); however, some files would benefit from being distributed through the Attic data network, due to their size or reuse frequency.

Support for moving files from gLite to ADICS/Attic was implemented in prototype form in the EU FP7 EDGeS project by extending the EDGeS Compute Element (CE) with additional code. In EDGI, the CE component of the gLite $\rightarrow$ DG bridge was further extended to support file references and improve data handling [123]. These enhancements make moving files to the Attic network much easier, more integrated, generic, and seamless than they were in EDGeS. In addition, the CE component used in the EDGeS solution was replaced with the more improved CREAM CE [124] component in EDGI.

The integration of Attic P2P data network and the gLite $\rightarrow$ DG bridge involved the following tasks to be performed:

- preparation of gLite jobs (shown as gLite UI in Figure 5.7)
- extension of CREAM CE component
- modification of the 3G Bridge
- modification of DC-API
- proper setup of the targeted BOINC project

5.2.1.1 Preparation of gLite Jobs

In order to make use of Attic in BOINC projects with the gLite $\rightarrow$ DG scenario, a few requirements must be met. In addition, users need to set up their jobs to fulfill some requirements.

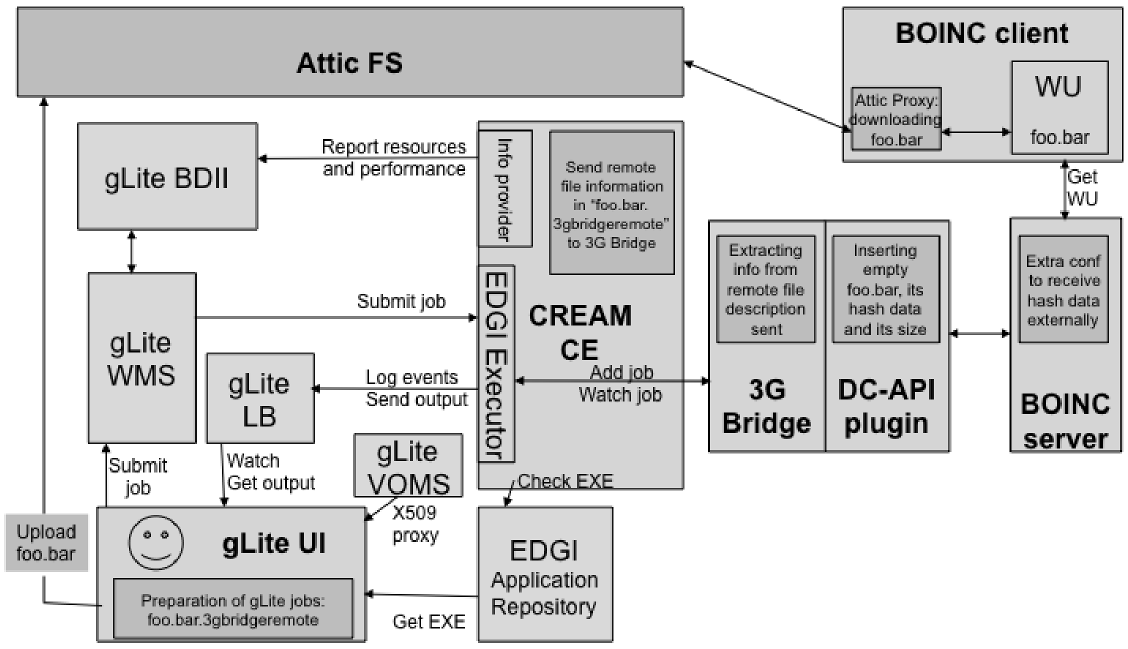
One requirement is that the targeted BOINC project must be set up properly to support handling Attic-featured workunits. Second, clients of the targeted BOINC project must attach to an Attic-proxy project as described in §5.3.

[4]The “3G Bridge” is the name of the software toolkit that enables migration of Service Grid jobs to Desktop Grid infrastructure.

A user (see Figure 5.7) must perform following tasks in order to use Attic with a BOINC project:

- upload required files into Attic using the `attic-deploy` Command Line Interface (CLI) tool, which is a wrapper for `curl` and sends a file to an Attic `DataSeed`, which, as mentioned previously, is a specialized Data Center (cacher) in the Attic network that can serve as an initial staging point for files, and
- for every file published to Attic, modify the job descriptor to reference the remote (i.e., Attic) files instead of the original input so the gLite → DG bridge can handle these files.

If a user has an input file called *foo.bar* to publish in Attic and use in BOINC, the following file has to be submitted with the job instead of *foo.bar*: *foo.bar.3gbridgeremove*. The file should contain the following information, each in a new line:



Attic FS
BOINC client
Attic Proxy: downloading foo.bar
WU
foo.bar
Get WU
gLite BDII
Report resources and performance
Info provider
Send remote file information in "foo.bar. 3gbridgeremote" to 3G Bridge
EDGI Executor
CREAM CE
Add job
Watch job
Submit job
gLite WMS
gLite LB
Log events
Send output
Extracting info from remote file description sent
3G Bridge
Inserting empty foo.bar, its hash data and its size
DC-API plugin
Extra conf to receive hash data externally
BOINC server
Submit job
Watch
Get output
gLite VOMS
Check EXE
X509 proxy
Upload foo.bar
gLite UI
Preparation of gLite jobs: foo.bar.3gbridgeremote
Get EXE
EDGI Application Repository

Figure 5.7: Required Modifications for gLite → DG Job Execution.

- the remote URL of the file (for example the Attic URL, or the HTTP URL of the file). In the case of Attic, this URL is returned from the `attic-deploy` script,

- the MD5 hash of the uploaded file, which can either be determined by using `md5sum` or by giving attic-deploy an optional argument to output the md5sum, and
- the size of the file (in bytes). This is an optional parameter, the value of which can be determined by using the `du` command.

It follows from the above steps that the original contents of the already published files are not transferred through the gLite $\rightarrow$ DG bridge. Instead, only one small (metadata) file is transferred.

In this case, the EDGI CE component of the gLite $\rightarrow$ DG bridge has only one task related to the *.3gbridgeremote* files: the contents of the file should be parsed and sent to the target 3G Bridge as a file reference.

5.2.1.2 Extension of the CREAM CE Component

When the CREAM CE component (see Figure 5.7) receives a job, it reads its input file list based on gLite job objects. Every input file has the following properties in the job object: the name of the input file and the GridFTP URL of the file. The CREAM CE fetches all the input files and places them under a storage accessible through HTTP(s). The same file content is persisted only once; therefore, if a user submits jobs with the same input file (e.g., 1000 times), the given file is saved only once on the storage. As soon as an incoming job's files have been fetched by the CREAM CE, the job is sent to the target 3G Bridge service, using *references* to input files, not the real file contents. A file reference includes the following information: name of the file, URL of the file (i.e., location on the CREAM storage), MD5 hash of the file, and size of the file. The URL, MD5 and size are (normally) calculated by the CREAM CE.

However, gLite users may pass Attic references through the system. Let us assume a user has a job with an input file called `foo.bar`, and has uploaded this file to Attic. Thus, the user knows the Attic URL of the file, the MD5 hash of the file, and the size of the file. In this case if a user doesn't want to push the original contents of file `foo.bar` into gLite, he/she has the possibility instead to send an Attic reference. In order to do so, he/she has to replace the file `foo.bar` with `foo.bar.3gbridgeremote` in the Job Description Language (JDL) and place the following information into `foo.bar.3gbridgeremote`: Attic URL of the file, MD5 hash of the file, and size of the file in bytes.

Once the CREAM CE detects a file ending with the *.3gbridgeremote* suffix among the input files, it parses the file's contents and uses the information stored in the file as the file reference to the target 3G Bridge service. To be precise, given that a file called `foo.bar.3gbridgeremote` is specified in the JDL, the CREAM CE will send an input file reference called `foo.bar` to the target 3G Bridge service, using the additional metadata stored in the transferred file (i.e., Attic URL, MD5 hash, and file size).

5.2.1.3 Modification of 3G Bridge

To achieve the above integration, the 3G Bridge component had to be modified to handle remote file references. This entailed both the Web service interface and the database schema being modified to enable the sending and storing of MD5 hash and file-size information.

The DC-API-Single plug-in was also modified to manage files published through Attic. To detect Attic files, each file of a job passing through the bridge is checked to see if the file's URL prefix (i.e., protocol) begins with `attic://` or `attics://`. If so identified, the 3G Bridge considers the given file as published within Attic (a fairly safe assumption) and adds the file (e.g., an empty `foo.bar`) to the workunit using a modified DC-API function call. This modified DC-API function receives the original file's MD5 hash and size as stored in the remote file reference sent by the modified CREAM CE.

5.2.1.4 BOINC Project Setup

In order to make the Attic integration fully functional with BOINC, one additional configuration must be done on the targeted BOINC project server. This modification is that the BOINC project must be set up to use the MD5 hash information stored by the DC-API function called by the 3G Bridge. For this, the <*cache_md5_info/*> -tag must be added in the BOINC project's configuration file within the <*config*> -section. This will make BOINC use the MD5 hash information stored by the DC-API function (instead of calculating it) for the added file. By injecting the MD5 hash into BOINC, the 3G Bridge is able to guarantee that BOINC uses the original MD5 of the file (as sent by the gLite user) instead of the calculated value, which would be incorrect, as the calculated value would reflect the hash of the place-holder file instead of the actual data.

Using the above mechanism, one is able to trick the BOINC project into using the (remotely) published files' MD5 hash and size information in the same manner as a local file.

Similarly, on the client side, the MD5 hash and file size information is checked only after the given file has already been fetched from Attic with the help of the Attic-proxy application (see §5.3).

5.2.1.5 Modification of DC-API

There were two limitations that had to be eliminated in order to make DC-API fit to the new requirements imposed by the 3G Bridge $\longleftrightarrow$ Attic integration.

1. Under the Attic BOINC Proxy project, an HTTP redirection for the files that are uploaded to Attic takes place. To differentiate among normal BOINC inputs and Attic inputs, Attic requires a predefined format of the physical name of the file registered under BOINC. Since the 3G Bridge handles workunits and their inputs through the DC-API interface and this interface generates the name of the inputs according to its internal rules, DC-API had to be modified to let the name of the file be defined externally. That is, the 3G Bridge had to be modified to support preservation of file names.

2. In BOINC, MD5 hash values are generated on the BOINC server for all input files and then checked on the clients after download. Since under the modifications to support Attic a proxy-file is sent to BOINC, this checking would fail for the downloaded file if comparing hashes from the original input. To solve this issue, the DC-API had to be modified to let the MD5 hash value be defined externally and passed into BOINC. This works provided the target BOINC project is configured appropriately, as shown in "BOINC project setup."

The original method of DC-API for inserting input files for a workunit is as follows:

Listing 5.3: Original Method to Add a Workunit with the DC-API

```c
int DC_addWUInput
(
    DC_Workunit *wu,
    const char *logicalFileName,
    const char *URL, DC_FileMode fileMode
);
```

To eliminate the limitations mentioned above, this was extended with a variable argument list, making it possible to define the MD5 hash and size information of files:

Listing 5.4: Modified Method to Add a Workunit with the DC-API

```c
int DC_addWUInputAdvanced
(
    DC_Workunit *wu,
    const char *logicalFileName,
    const char *URL, DC_FileMode fileMode,
    const char *md5hash,
    const char *fileSize
);
```

If the `fileMode` argument's value is `DC_FILE_REMOTE`, the function assumes two additional arguments: the MD5 hash (i.e., *md5hash*) of the file and the size of the file in bytes (i.e., *fileSize*). By using the extended method, the bridge is able to define, and pass on to BOINC, the name of the input file, the MD5 hash, and the file size.

5.2.1.6 Example Job Submission

The use-case given here assumes a user has an application using an input file called *foo.bar*, with the following JDL:

Listing 5.5: Original gLite JDL Specification

```
  Executable = "samplejob";
  InputSandbox = {"samplejob", "foo.bar"};
  OutputSandbox = {"samplejob.out"};
```

First, the user calculates the MD5 hash (using some tool, for example the Linux `md5sum` command) and size of the file. For example, an MD5 hash might be `b87e8e01fb079a938640c6646242c20a`, and the size could be `9053342` bytes. The user then uploads the file to Attic and receives an `attic://` URL as output, for example `attic://example.org/dl/meta/pointer/1234567890`. Next, the user creates a file called *foo.bar.3gbridgeremote* with the following contents:

```
attic://example.org/dl/meta/pointer/1234567890
b87e8e01fb079a938640c6646242c20a
9053342
```

The `foo.bar` reference within the JDL is then replaced by a reference to `foo.bar.3gbridgeremote` (thus, not the original file's content, but a small metadata file is passed through the bridge):

Listing 5.6: Modified gLite JDL Specification with Reference to Attic Metadata

```
Executable = "samplejob";
InputSandbox = {"samplejob", "foo.bar.3gbridgeremote"};
OutputSandbox = {"samplejob.out"};
```

At this point the job can be submitted. All the specified (i.e., in the JDL) input files are uploaded to the gLite Workload Management System (WMS). The WMS selects the CREAM CE for execution and sends the job onward for processing. At this point, the CREAM CE fetches the job's input files. In case of this particular job, it identifies a remote file reference in the form of *foo.bar.3gbridgeremote*. The bridge reads the file's remote (Attic) URL, MD5 hash and size from the stored metadata. The CE then sends this information to the selected 3G Bridge service, using the filename without the `3gbridgeremote` suffix, but with the other metadata. For example, in the above scenario, it would use filename *foo.bar*, URL `attic://example.org/dl/meta/pointer/1234567890`, MD5 hash `b87e8e01fb079a938640c6646242c20a` and size `9053342`.

Once the 3G Bridge has received the job, it calls the DC-API plug-in to submit the job to BOINC. The DC-API plug-in uses the extended `DC_addWUInput` function to add the remote file to the BOINC workunit. The extended DC-API function realizes that the file reference is an Attic file reference, and operates as follows:

- adds the file *foo.bar* to the workunit,
- creates an empty file called `1234567890` within the BOINC project server's download directory,

- assigns the URL of the `1234567890` file to the *foo.bar* input, and
- assigns the MD5 hash `b87e8e01fb079a938640c6646242c20a` to the *foo.bar* input,
- assigns the size `9053342` to the *foo.bar* input.

From this point on, once a client fetches the created workunit, it is the task of the Attic Proxy application (running on the client machine) to fetch the file. This will be done not from the BOINC project servers, but rather from the distributed Attic data sharing network.

5.2.2 Data Transition from ARC $\rightarrow$ DG

The way data is handled in NorduGrid, and subsequentially with the Advanced Resource Connector (ARC), differs from the other major Service Grid middleware. In ARC, data transfers are centralized at each participating site, rather than hosting a centrally managed infrastructure such as gLite's SRM. This is a result of ARC being specifically designed to be as non-intrusive as possible. A key requirement for ARC was that a site deploying ARC should not have to install any software on the nodes. A related requirement was that nodes might not even have network access outside the cluster. This means that the ARC server at a site works as a gateway that must take care of all data transfers related to the Grid. For performance and fault tolerance, the data transfer service (called up- and downloaders) can be split from the ARC server and replicated across several machines. These nodes can then be tailored for fast I/O and network. This approach also has the advantage of providing a site-wide cache and not tying down cluster nodes waiting for off-site file transfers.

The ARC up- and downloaders support a wide range of standard internet protocols (e.g., FTP and HTTP), standard grid protocols (e.g., GSIFTP/GridFTP), and ARC-specific protocols such as Chelonia. A full description of ARC and it's supported data protocols can be found at `http://www.nordugrid.org/manuals.html`.

5.2.2.1 Data Transition to Desktop Grids

For the purposes of completeness, three possible scenarios for data transfer from the ARC environment to Desktop Grid clients are identified. *Scenario 1* regards files stored in authenticated storage inside the ARC system; *Scenario 2* refers to files which are already made available in an open manner and could be directly downloaded by DG clients; and *Scenario 3* deals with the case of files that would benefit from Attic file distribution due either to their size or use frequency.

Please note that for a detailed explanation of the ARC Bridge and its implementation, see [123]. The text here is designed to be an overview of the different data scenarios and to give an introduction to the Attic implementation that can be used as a reference and comparison for the other Service Grid systems in EDGI that use Attic.

Three different data transfer paths for ARC $\rightarrow$ DG were identified as needed for the ARC Bridge implementation, and are as follows:

Scenario 1: Data in authenticated storage The first scenario concerns data that comes directly from a client or data available only on ARC-specific infrastructure such as Chelonia, or that which requires authentication to access, such as GSIFTP. This data is downloaded by the ARC Bridge (which has credentials to access the data) and then is published on a local Web server, making the files available through HTTP. The locations of these files, along with associated metadata, are then transferred to the 3G Bridge. After transferral, the Desktop Grid clients can directly download the input files from the HTTP server associated with the ARC Bridge. Note that this situation is centrally serving data, in a manner very similar to a BOINC server.

Scenario 2: Data already available somewhere for DG clients The second path is for data that is already available through a standard protocol (e.g., HTTP or FTP) and does not require credentials to access. In this case the ARC Bridge will not stage the data locally, but will transfer only the URL and metadata to the 3G Bridge. Desktop clients can then directly download the data from the original source.

Scenario 3: Data benefiting from Attic hosting The final path for transferring data to DG clients relates to Attic files. For files that are frequently used, it makes sense to stage them to Attic, so that multiple clients can download them and the Attic network manages the load and scalability factors involved in the distribution. Uploading files to Attic is considered out-of-scope of the ARC and 3G Bridges, since the user knows best if a given file is frequently used and large enough that it should be distributed with Attic. It should be noted that the ARC bridge could try to make this determination, based upon historical examinations of files or executables, or through some other Artificial Intelligence (AI) mechanisms, but the cost/benefits of doing so have yet to be evaluated.

For the moment, as in the gLite implementation, Attic file-uploading is left to the end-user or his/her Virtual Organization (VO). Note that the file catalogs used by ARC VOs are already capable of storing Attic URLs as alternates, and therefore can be used for this purpose. In this scenario, the ARC Bridge works similarly as in the second path, functioning as a pass-through for the Attic URL. The Desktop Grid client will then retrieve the data from Attic after the 3G Bridge makes the appropriate modifications to the workunit to dereference the file.

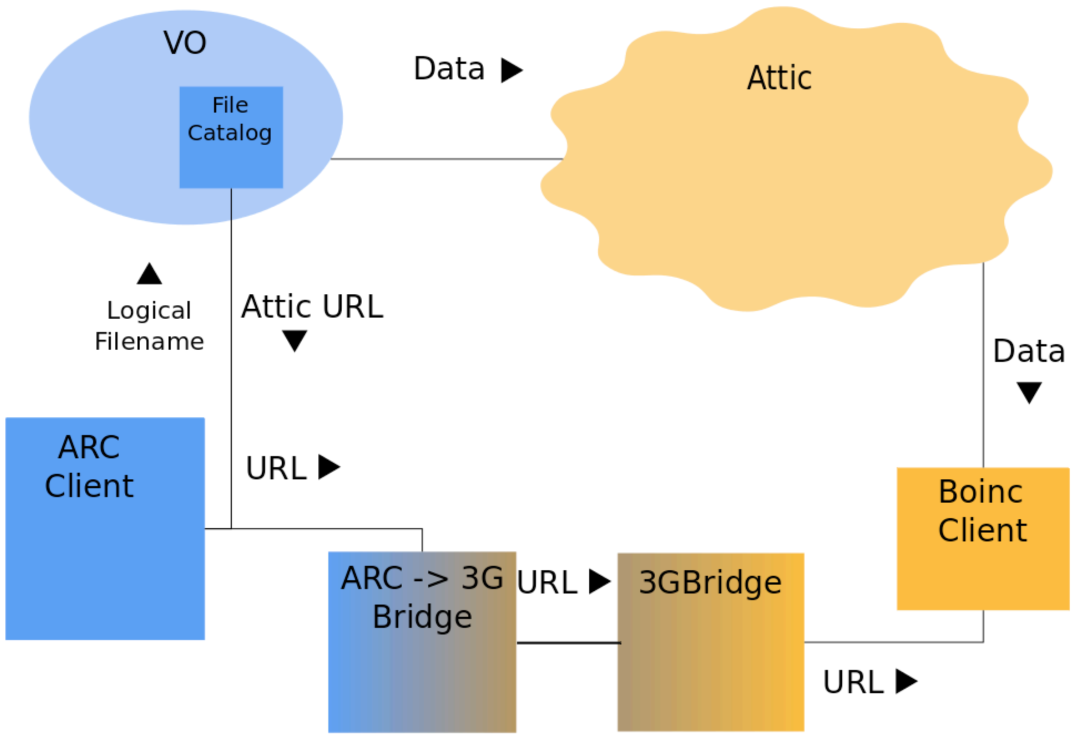


Figure 5.8: Transitioning Data Through the ARC Bridge to Attic — shows how data is first published to Attic, then the file reference location (URL) and MD5 hash as passed through the 3G bridge.

The new features developed by EDGI (i.e., the Attic support and the support for bypassing data-staging on the ARC Bridge) can be used seamlessly by an ARC client. In addition to specifying a URL, it is necessary to specify the MD5 sum and (optionally) the size of the file in question. This is handled by appending options to the URL in the standard ARC fashion described in: `http://www.nordugrid.org/documents/URLs.pdf`.

Below are examples of the four possible ways of specifying an input file.

- A local file:
  `(inputFiles=(‘‘foo.bar’’ ‘‘’’))`

  → Use the *foo.bar* file from the client's current working directory and stage it through the bridge.

- A Grid file:
  `(inputFiles=(‘‘foo.bar’’ ‘‘gsiftp://example.org/foo.bar’’))`

  → Retrieve the file from a NorduGrid storage element and stage it through the bridge.

- A Grid file on a public server:
  `(inputFiles=(‘‘foo.bar’’ ‘‘http://example.org/public/foo.bar; md5=b87e8e01fb079a938640c6646242c20a:size=9053342’’))`

  → Do not stage the file through the bridge, but let the clients retrieve the file directly.

- An Attic file:
  `(inputFiles=(‘‘foo.bar’’ ‘‘attic://example.org/dl/meta/pointer/ 1234567890; md5=b87e8e01fb079a938640c6646242c20a:size=9053342’’))`

  → The Attic file bypasses the staging; instead it is retrieved directly by the clients as in the preceding example.

Figure 5.8 shows the general path as data moves through the network. Note how the input file itself is uploaded directly to Attic before the job is submitted to the 3G Bridge, and only the file reference and associated metadata are passed through the 3G Bridge. This architecture is very similar to that provided for the gLite implementation (see §5.2.1) and puts the least amount of stress on the bridge, while retaining file distribution decision-making power with a client[5] (in

[5]File uploading is handled using the Attic CLI program (or associated libraries) that allows users to pre-stage the files to the Attic network before submitting dependent jobs.

this case ARC), who has better knowledge of whether or not a file should be distributed to the Attic system. For a more detailed discussion of data transfer with ARC through the 3G Bridge, see [123].

5.2.3 Data Transition from UNICORE $\rightarrow$ DG

The last Service Grid infrastructure included in EDGI is that of Uniform Interface to Computing Resources (UNICORE). As in the preceding sections, the text here seeks to describe the basic architecture and show how it relates to Attic. Figure 5.9 gives an overview of the UNICORE data architecture, which is divided across three layers: *client*, *service*, and *system*. Users and their applications can access UNICORE via different client tools and the *UNICORE System's Services*. The UNICORE System's Services, associated with the service layer, are loosely coupled Java Web services registered in typically one well-known Service Registry. The fundamental services offered are the following:

- *A XNJS-Service*, which embeds a Compute Resource (e.g., a cluster) inside the UNICORE system. This allows a user access to compute a job.
- *The Storage Management Service*, which embeds a global (system-wide) Storage Resource. A user can upload large or often-used input files to this global storage. After job execution a user can download the job's results from the global storage. The Storage Management Service is a so-called "Atomic Service" inside UNICORE and operates independently of other core services in a Web-Service based implementation.

Beside the components shown in Figure 5.9, there are several other services, such as the *Workflow Orchestration*, *Common Information Service*, and the *UNICORE Virtual Organization Service* for querying a VOMS service. One or more services can be running inside a hosting environment. Every inside connection to a service has to pass an access policy-enforcing gateway. The residual layer (the system layer) represents a concrete system or resource, e.g., a cluster. The associated local resource management system controlling the cluster is integrated into UNICORE via the Target System Interface. In addition, a local resource or cluster typically has it's own storage (see §2.4.1).

UNICORE internally supports BFT transfer, OGSA ByteIO, UDT, HTTP and GridFTP storage access for downloading input files and uploading output data.

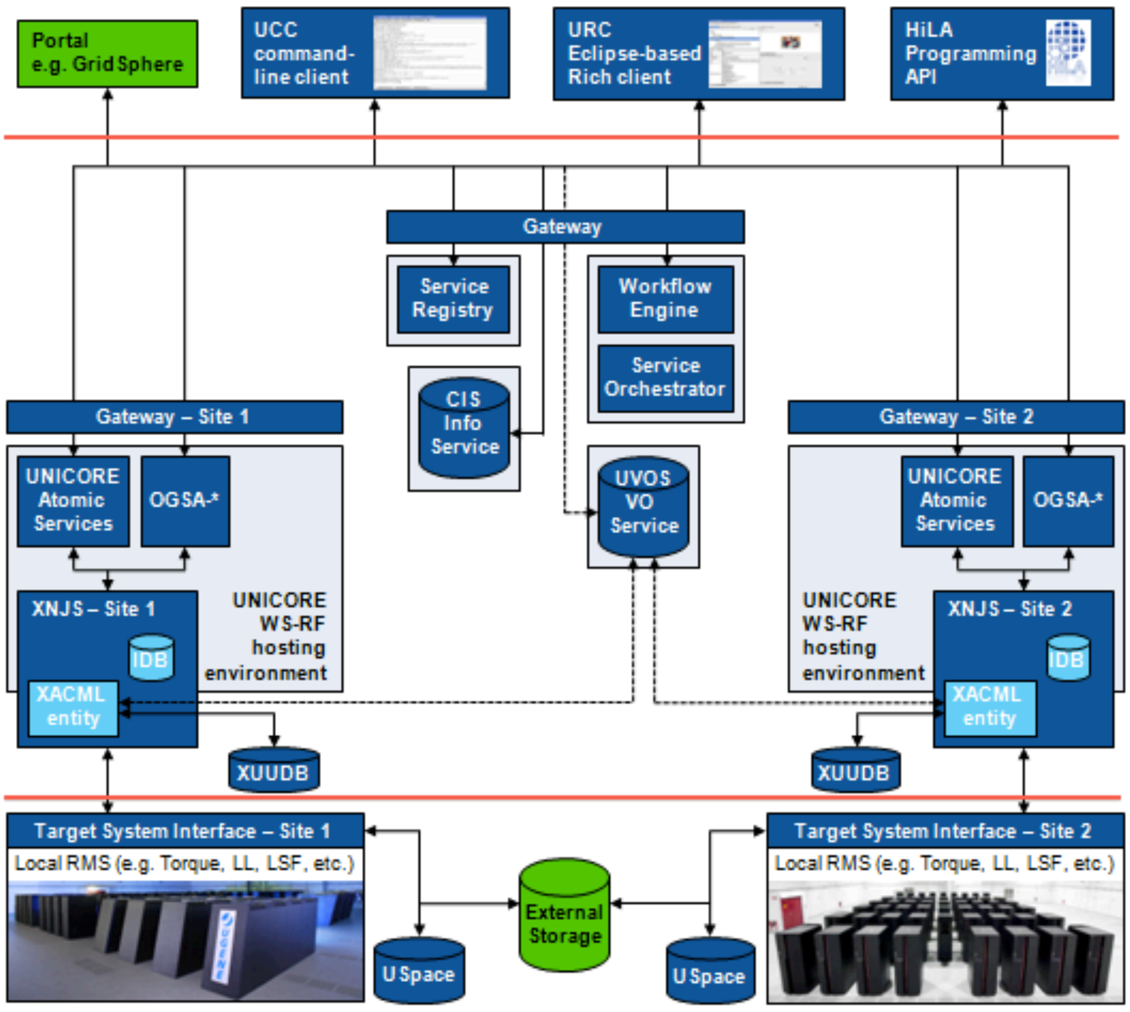


Figure 5.9: UNICORE Architecture — shows how UNICORE is segregate into client, service, and system layers (solid horizontal lines).

The *Storage Management Service*, a Java Web Service, is an abstract filesystem-like view on a storage resource and therefore is an adapter between the UNICORE System and the real storage. Originally used for local or cluster shared filesystem, it is currently used to integrate the Hadoop Distributed File System [125] into UNICORE clusters. The UNICORE storage adapter is built with extension in mind. Therefore, API-level functionality is reduced to a small subset of primitive functions that must be implemented for every type of storage.

5.2.3.1 Potential Data Transition Paths

During the analysis of ways to move UNICORE input files (or their references) through the EDGI bridge and to make them available to DG clients, three transition paths were identified. This section will discuss the possibilities and efforts of each variant. The analysis considers the following aspects: required effort to realize solution, gains in functionality, and user usability factors (and acceptance).

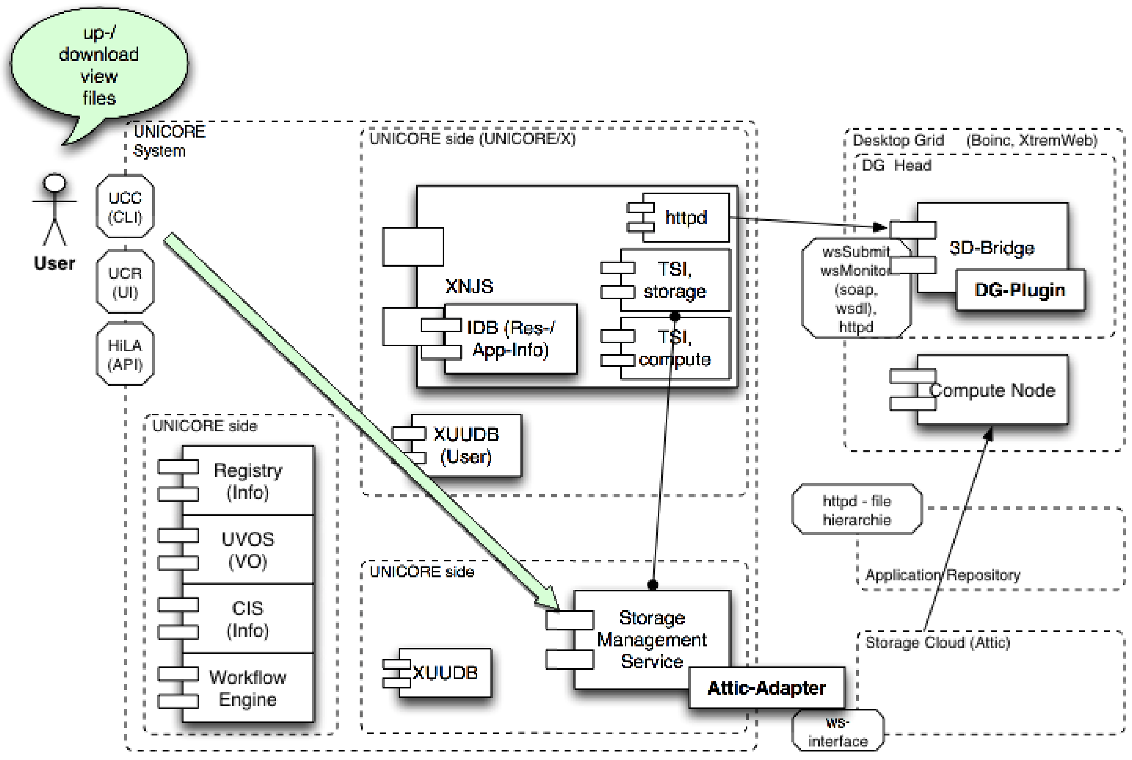


Figure 5.10: Combined EDGI and UNICORE Architecture — shows the overall architecture developed in EDGI consisting of UNICORE, an arbitrary Desktop Grid (accessed through the 3G Bridge), and a compact representation of the Application Repository and Attic. The green line shows a user uploading data to the UNICORE Global Storage (e.g., to Attic via an Attic-Adapter). The thin black lines show data flow during the process of job execution.

Separate system (Variant 1) The first option identified for moving files from UNICORE to Desktop Grids is to implement a system similar to that employed by gLite and ARC. In this type of system, publication of files is out-of-bounds, the resulting file locations (i.e., URLs) and associated metadata (e.g., MD5 hash and file size) are passed to the 3G Bridge to be processed and populated into DG workunits.

For this to take place, users, scripts, or applications would use the Attic command line interface (CLI) to publish data to the Attic network. The resulting `attic://` URL would be included by UNICORE in the job description (i.e., JDL) that is sent to the 3G Bridge.

The advantages of this situation are that it decouples publication from the bridge and keeps the decision-making process close to the users as to what files should be distributed with Attic. Many files are not suitable for Attic and it is desirable and efficient to use Attic only to distribute files that are larger and/or are required by multiple clients. It is difficult to automatically ascertain if a given file should be sent to Attic, or simply hosted on a project Web server. Also, since publishing files off-line is the path employed by both the gLite and ARC implementations, it would not require a new, UNICORE-specific, Attic plug-in to achieve UNICORE/Attic integration.

The disadvantage of this approach is the same as with gLite and ARC. It requires the user (or the user's agent) to decide which files to publish. Since this path maintains that the user must specify the files that are multi-use, public, and should be moved to Attic, it requires a certain degree of user education to put into production.

Subcomponent of UNICORE (Variant 2) UNICORE's global storage systems are accessible via the Storage Management Service, which can be seen as an adapter to arbitrary file-like storages. An example user scenario using an adapter is shown in Figure 5.10. This would be a natural integration of Attic for end users and would not require them to modify their behavior, as the same UNICORE commands and interfaces could be used to publish files to Attic. By using the default UNICORE storage mechanism, the Attic-Storage becomes a part of the UNICORE-Ecosystem. Thus a UNICORE user can access Attic storage in a manner similar to any other type of (UNICORE) storage and isn't forced to learn something new or adapt scripts or applications. The DG-side will get the native Attic-url; thus a DG client can access Attic directly.

To achieve this solution, which would allow UNICORE users to push files to Attic through their standard interfaces, two classes have to be implemented as shown in Figure 5.11. Additionally, a UNICORE-specific `DataSeed` host would have to be implemented to pre-stage the files to the Attic network. The parameters of the methods expect Java byte-streams, which are provided by the Attic library for client-downloading.

The advantage of this approach is the seamless integration for end-users, who do not need to learn the Attic-specific CLIs used common to ARC and gLite. However, the disadvantages are that it requires extra development effort, creates a UNICORE-specific solution, and also requires the creation of

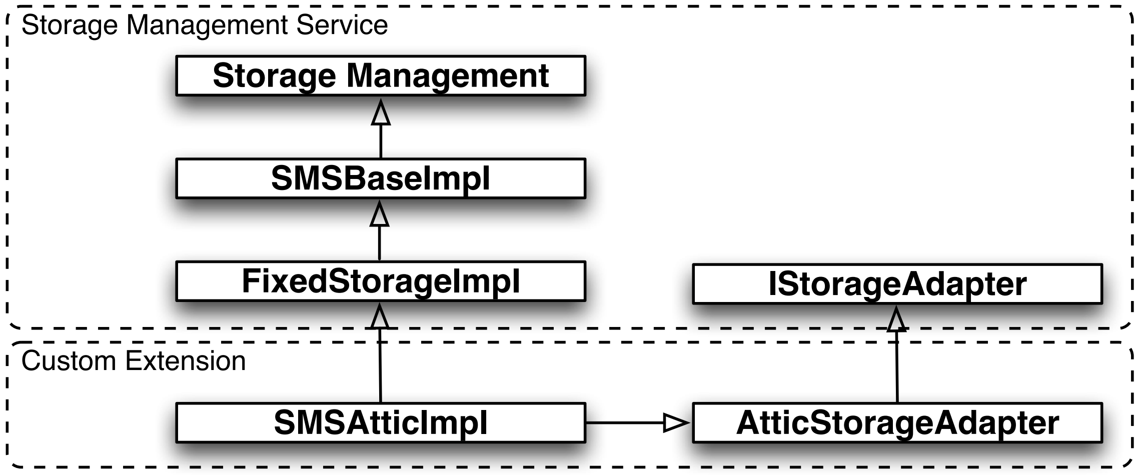


Figure 5.11: UNICORE Storage Adapter Implementation — shows a rough class diagram in which the two classes at the lower end contain the adapter functionality of a UNICORE Storage Service.

both publication and retrieval mechanisms within the UNICORE-Ecosystem, since both directions are required for implementation of the UNICORE file API.

Semiautomatic System (Variant 3) The last evaluated data transition path is an extension to *Variant 2* and solves an additional issue: UNICORE users are typically from non-computer science disciplines and don't understand the technical details and steps involved in file publication and retrieval. In some cases a user may not have enough knowledge to decide correctly whether or not a file can be sent to special purpose storage (like Attic) instead of UNICORE storage. In this case, the users would likely ignore Attic, which would result in a degradation of the overall performance of the targeted EDGI-solution.

As shown in Figure 5.10, the storage-TSI is responsible for delivering data to the Desktop Grids. This component could be extended, so that "appropriate" files are not handed over via HTTP, but instead via Attic. The UNICORE Storage Mechanism would be able to handle the file transfer between two Storage Management Services (provided Attic adapters were implemented).

In the case that users are aware of the performance degradation of not using Attic, they can (and should) opt to use Attic separately. However, despite being semi-automatic by trying to determine which files "should" be sent to Attic, this solution is limited by the intelligence of the AI making these decisions. An evaluation would be need as to whether it is possible to programmatically define what “appropriate files” are. To this end, systems such as disk caches and history counters could be explored.

5.2.3.2 Resulting Implementation

Variant 1 proved itself the easiest to deploy, and kept the implementation in-line with the system that has been developed for gLite and ARC. Although it requires that the users, or some other intermediary program, to make an explicit decision to publish files to Attic, it is still a useful system that can prove beneficial for Desktop Grid jobs. Creating an Attic Adapter for the UNICORE Storage Service remains an option for future exploration; however, it would require additional development efforts that are beyond the scope of this research.

5.3 BOINC Integration

BOINC data distribution, as seen in Figure 5.12, and detailed in §2.3.1, is currently provided through centralized HTTP Web servers. To provide redundancy and scalability, BOINC projects *mirror* this data to many servers around the world. This system works fairly well for BOINC projects. However, it has the disadvantage of requiring centralized project servers to distribute vast amounts of project data, leading to additional costs both in terms of hardware and network capacity, as well as support technicians. Furthermore, projects such as EDGeS and EDGI, in which data can move from Service Grids to a volunteer BOINC Desktop Grid, it can be problematic to host these large data sets, since there is no central “project” to which they belong (because they come from many different users). A solution that leverages existing client-side network capabilities to share data can be advantageous in both of these scenarios.

The Attic software described earlier in this chapter, along with the Service Grid enhancements and 3G Bridge translation software, provides a dynamic data sharing network for BOINC that is more flexible than BOINC’s current static server setup. In Attic, data is first published to a P2P network layer consisting of volunteer hosts that opted to donate networking and storage resources to the project. These resources can be either typical volunteer PCs, or a set of known resources, if the project’s security policy dictates such an action. In either case, once the data has been published, it *self propagates* through the network until a certain (optionally pre-defined) maximum replication count is reached. The replication number is defined at publish time by the data publishing agent (e.g., as a command-line parameter).

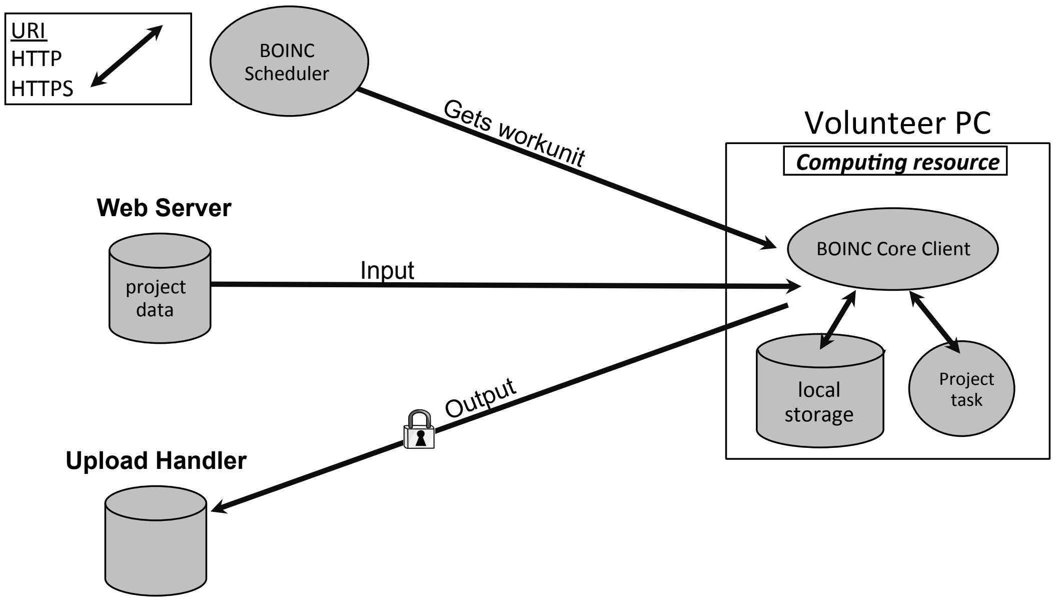


Figure 5.12: Current BOINC Download Workflow — depicts current BOINC download practices, where input data is downloaded directly from a static list of HTTP mirrors.

When publishing a file to Attic, the data publishing agent will specify the location of a “seed” `DataCenter` (e.g., a `DataSeed` or other hosted environment) that has the initial copy of the data. The seed location will be registered with the `DataLookup` Service as an initial replica, with the caveat that it can be optionally de-prioritized once more replicas are on the network, to help prevent too much network load being put upon the seed. Once a publication is complete, a unique URI is returned to the publisher that references this piece of data. This URI can then be used by BOINC clients to retrieve the file from replica hosts (i.e. additional Data Centers that have cached the file). Specifically, for the BOINC client, this is done seamlessly through the local Attic Proxy project that runs on the client machine, as described in §5.3.

Attic Proxy

The **Attic Proxy** [126] application is designed to allow the Berkeley Open Infrastructure for Networked Computing (BOINC) (see §2.3.1) access to the Attic system. To avoid patches to or a re-write of the BOINC Core Client, which would cause many incompatibilities with older clients and projects, the Attic Proxy was written. This application sits in the background on the computer doing the processing, i.e., the client. The recommended way to distribute the application is to create a second BOINC project specifically to host the Attic Proxy. This will have the `non_cpu_intensive` flag set, so that when the BOINC client connects, it knows that it should run the application while ignoring any rules on CPU and memory usage. After the Attic Proxy project is attached, any scientific project can use it to access files hosted on Attic. Figure 5.13 shows how the Attic Proxy integrates with the BOINC server and client machines.

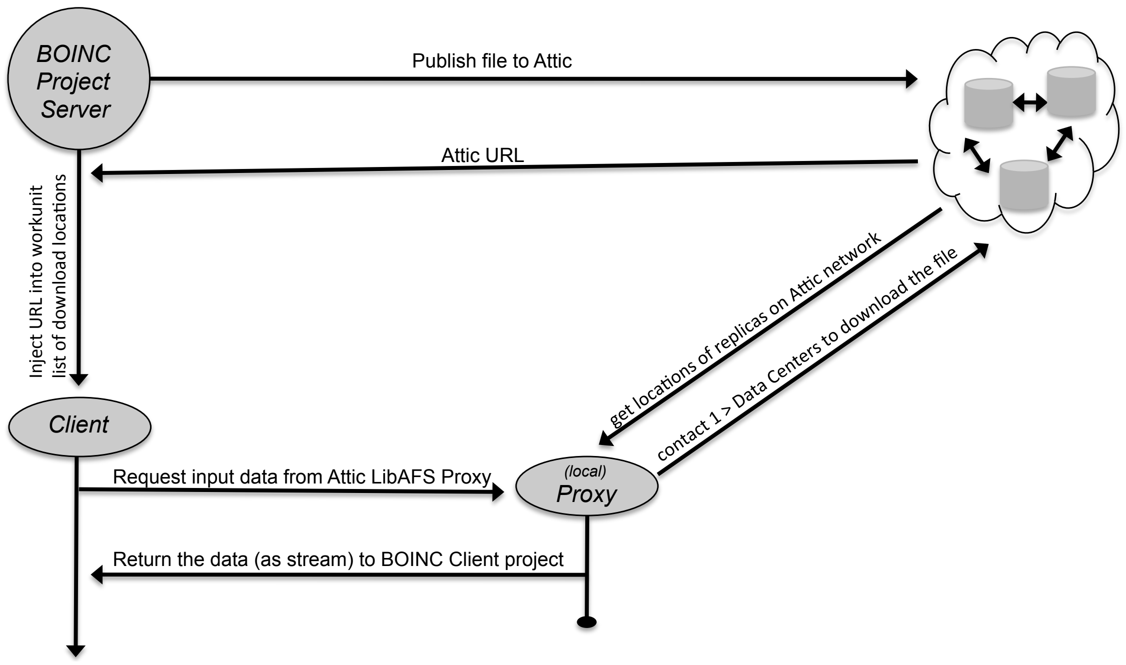

BOINC Project Server
Publish file to Attic
Attic URL
Inject URL into workunit list of download locations
get locations of replicas on Attic network
contact 1 > Data Centers to download the file
Client
Request input data from Attic LibAFS Proxy
(local) Proxy
Return the data (as stream) to BOINC Client project

Figure 5.13: BOINC Downloading Using Attic — shows a simplified depiction of how a BOINC application could use Attic to download data using the local BOINC proxy application discussed in §5.3.

To provide the “local proxy” shown in the figure, Attic Proxy runs a limited embedded Web server on (by default) port 23456. When a request for a file is sent to this server, the ID is extracted from the URI and is used to retrieve the file from Attic. Once the file is retrieved, it is sent using the still-open HTTP connection. From a client's point of view, it simply appears as if the Web server is taking a long time to respond.

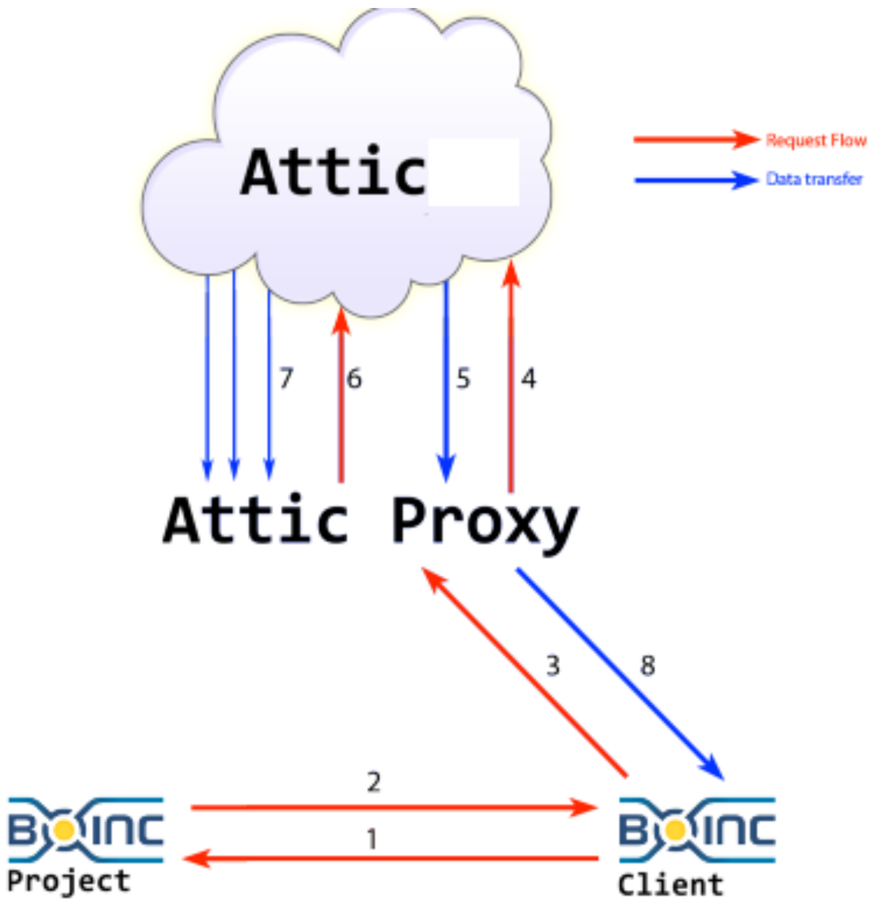


Figure 5.14: Attic Proxy Workflow — shows the workflow for downloading a file in BOINC using the Attic Proxy.

The process of downloading from Attic, as shown in Figure 5.14, is as follows:

1. BOINC client requests a file containing a UUID from the project.
2. The project detects the UUID in the request and redirects to the Attic Proxy running on the client.
3. Attic Proxy receives a request for a file from the BOINC middleware, which has a reference to a (local) URL the proxy handles.

4. The pointer collection (as XML or JSON) is retrieved from the `DataLookup` Service, a remote REST-based Web service that stores caching locations and metadata about the Attic network.
5. The Pointer Collection is returned.
6. Requests are made to Attic Data Centers (i.e. Endpoints) for the various chunks of the file.
   - If a chunk's MD5 hash does not match the expected hash, the process is repeated from Step 6. If the max retries limit is reached, the process fails with a `404` error.
7. The downloaded chunks are downloading and reassembled into a single file. The MD5 hash of this file is checked. If it fails, Attic Proxy returns a `404` error.
8. The final file is sent back the BOINC client over the open HTTP connection.

In order for a BOINC client to retrieve files via the Attic Proxy, a redirect rule must be defined for the project's download folder that detects when a request is made for a file whose name is a UUID. This redirect points to `http://localhost:23456/data/<<file>>`. As the Attic Proxy runs on a PC hosting a BOINC client, it is likely that this system will be carrying out many CPU and memory intensive tasks. With this in mind, steps were taken to reduce the footprint of the application.

To keep memory usage to a minimum, downloaded chunks (individual pieces of a larger file, that can later be reassembled) are stored as files instead of in memory. This is also useful for larger files, which could cause much page swapping if stored in memory. The download processes (i.e., threads) are set to a lower priority than normal, allowing them to yield to other tasks. Any non-time-essential loops have extra `Sleep(1)` calls in them to facilitate this yielding more easily.

The Attic Proxy is written in ANSI C to ensure that it does not add any additional requirements to BOINC clients (as seen in the requirements specified in §3.4.4). It uses the libXML2 library for parsing XML, libCurl for HTTP client features, and the Mongoose-embedded Web server for serving requests. Attic Proxy has been tested and has been found to work Windows XP or higher (32 and 64 bit), Mac OSX 10.4 or higher, and Linux.

5.4 XtremWeb Integration

The XtremWeb platform (XWHEP) (see §2.3.2) addresses data access by two meanings. First it implements data access using Uniform Resource Identifier (URI) [127]. Describing URIs in detail is out of scope of this document, but the reader may at least be aware that URIs are standardized data locators, each of which is mainly composed of a *schema*, a *server address*, and eventually a *path*. The schema defines the protocol used; the server address defines the location over the Internet; and the path defines the location inside the server without giving a clue as to the storage technology effectively used. Some well known schema examples are *FTP* for file transfer protocol, and *HTTP* for the hyper text transfer protocol. It is the middleware (the server and the distributed clients) responsibility to use URI correctly and especially to interpret the schema correctly.

Proposing URI usage to access data, XWHEP allows any URI-capable client to access data, as well as referencing data from jobs. Any job managed by XWHEP may use data in this way, for example, for input files and results sets. All data is defined by URIs, enabling both inputs and results to be stored anywhere – independently of the location and technologies. XWHEP is therefore data agnostic. This is true for any data used from inside the platform.

XWHEP proposes its own URI schema: `xw://`. Any data may then be stored in an XWHEP data repository. If this is the case, the data URI schema is `xw` and the data transfer protocol is the one defined by XWHEP. Clients must have access to an implementation of the XWHEP transfer protocol.

Figure 5.15 shows data access in the XWHEP platform. Using the XWHEP, a user may access (i.e., insert, read, write, remove) data independently of the technology used as soon as the data is referred by URI. Also, a the client has access to an implementation of the protocol. The other part of the platform, the computing element (a.k.a., the worker), works the same way since it uses the same XWHEP communication library. When the worker downloads a job, it accesses the job's data and stores the job's results, which are then referenced through URIs.

Attic Protocol Handler

The XtremWeb API uses the `java.net.URL` object to access streams of data from arbitrary endpoints. A `URL` exposes an `InputStream` from which data can be read. Attic integrates with XtremWeb by registering new URL *protocol handler* with the XtremWeb system for the `attic://` and `attics://` URL schemes [128].

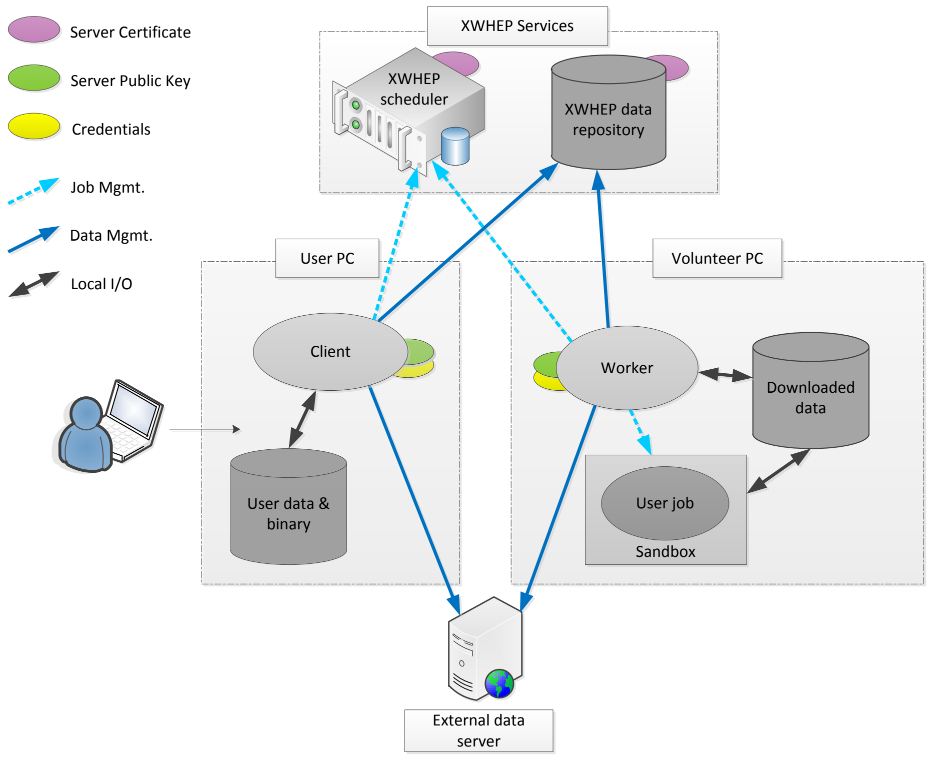


Figure 5.15: XWHEP Architecture — shows how data is accessed.

The first uses a non-secured TCP connection while the second uses mutually authenticated TLS over TCP. The Attic URL handler can be registered with any system by calling the `AtticProtocol.registerAttic()` static method. After registering, if an Attic-specified scheme is encountered, then the Java Virtual Machine will invoke the Attic URL handler. Since XtremWeb is a Java application that opens a connection to download any URIs for job input files, this integrates easily.

Once the Attic protocol handler is registered with XtremWeb, there is essentially no more integration necessary to enable XtremWeb to download Attic files. The Attic library acts as a “black-box” that, once referenced, provides the needed functionality for XtremWeb without requiring any further modification to its code beyond the few lines of registering the protocol handler. This is achieved by the Java Runtime Environment, which automatically loads the appropriate library to download the `attic://` file; XtremWeb processing continues as normal (i.e., as it would with a standard http or https URL).

Internally, the Attic URL handler creates a new `AtticURLConnection` to handle the actual streaming. The `AtticURLConnection` creates a stream source and a stream sink. The source pulls data from the remote endpoints using the downloading mechanism described in §5.1.5. Specifically, the stream source has a `DownloadTable` with a priority queue based on the sequence of chunks. Hence the chunks are more likely to be downloaded in order of their byte offsets. The stream sink receives the chunks from the source as and when they arrive. The sink is implemented as ajava.io.InputStream, specifically an instance of `AtticInputStream`, and is the input stream that the `URL` exposes to client code. Thus it has standard methods for reading bytes. The fact that the bytes are coming from different remote servers is transparent to client code.

The `AtticInputStream` uses a priority blocking queue to store chunks that arrive asynchronously from the stream source. When client code calls its `read` method, the input stream checks to determine if a stream is currently being read. If so, it attempts to read the client specified number of bytes from the current stream. Otherwise it looks in its queue for the next available stream and begins to read from that one. If the current stream does not have as many bytes available as were requested in the `read` method, then the next stream in the queue is used to complete the `read` request.

The stream source notifies the sink (asynchronously to the `reading` process instigated by the client code) when all chunks have been downloaded. When this occurs the sink places a poison marker stream in the queue so that, when it removes this stream from the queue, it knows that there are no more data available and it can return the correct value (-1 as specified by the `java.io.InputStream` API) to the client code.

If any chunks arrive out of order, despite the sequential prioritization of the `DownloadTable`, the input stream holds these temporarily until the missing chunks that sequentially precede it are inserted into the queue.

The streaming of data does not use any interim storage on the local machine. Rather, the stream source passes the handles of the remote data streams directly to the stream sink, i.e., the `AtticInputStream`. Configuration also allows for verification of data as it is downloaded. In this case the configured in-memory buffer must be large enough to hold a full chunk which can be verified using the MD5 hash in the metadata.

The above system allows XtremWeb to use Attic with little essential modification to its existing code (about three lines) and in a manner identical to how it handles FTP, HTTP, or other file transport protocols. Thus, XtremWeb integration is much simpler and native than the BOINC solution, yet provides no less functionality.

CHAPTER 6

Assessment

One important element of conducting successful research is assessing how it impacts the broader community and contributes to knowledge in the target field. A key element to applied science is gauging the applicability and performance of any new system and showing how it proves itself novel and ultimately useful. This chapter's goal is to evaluate the work presented in this dissertation. Two methods are used: first, seeing how ADICS impacted the Desktop Grid and related communities, and second testing the ADICS data management architecture. After presenting these two aspects to ADICS' evaluation, the chapter concludes by identifying areas in which the protocol and its resulting software implementation could be improved upon.

6.1 Community Impact

The research involved in the development of ADICS and its impact upon the scientific community is both qualitative and quantitative. Qualitatively, the fundamental question is: How has ADICS positively impacted its target communities and otherwise created value-added benefits and further research? Quantitative evaluation metrics further this discussion by seeking to explore how the improvements can be measured and compared against base-line functionality. For example, one quantitive metric would be if the use of ADICS provides a more useable, scalable, or higher-performance data distribution paradigm than the current state-of-the-art solutions.

Focusing on broader impact, I believe the research undertaken here has had an impact far beyond providing a novel thesis topic. First, the base research into ADICS' new data management approach led to the successful completion of the original grant that funded it. Second, the ideas that grew out of that original work led to a new data distribution architecture (ADICS, see Chapter 4), which was substantially involved in the creation and execution of two large multi-year European infrastructure projects (EDGeS and EDGI). Third, the development of ADICS and its ideas contributed to other research projects at Cardiff University. Lastly, through the collaborations and resources made available in EDGeS and EDGI, the ADICS data distribution paradigm was able to evolve into a useful and well-received software package (see Chapter 5) for data distribution both within Desktop Grids and also for data migration between Service and Desktop Grids.

For a further discussion of the projects impacted by this work, their details, and how they tie together both technically and chronologically, see Appendix B.

Perhaps the greatest demonstration of the influence of this research is that the developers of the world's largest Desktop Grid middleware, BOINC (see §2.3.1), have been supportive collaborators in the development of ADICS, and have shown an interest in integrating Attic into their core software. Modifications to BOINC have already taken place to enable Attic connectivity, with the addition of a new project flag that enables add-on projects such as the Attic Proxy (see §5.3) to run in the background as daemons. The ADICS infrastructure was also presented to the wider Desktop Grid community thorugh an invitational talk at the 7th BOINC Workshop in Hannover [129] and was well received. This acceptance by BOINC and dissemination into the BOINC community highlight the impact of the research, both *(1)* as a way of making peer-to-peer data distribution acceptable (by conforming to the requirements of the target community) within volunteer computing environments, and *(2)* as proof that the ADICS architecture has passed the high bar of community acceptance and been deemed a useful candidate for enhancing Desktop Grids.

Further integration with BOINC would have a massive impact on the use of Attic. However, such integration work is beyond the scope of the research presented here, and is left as a tantalizing future task.

6.2 Attic in Action

The prior section showed how ADICS has impacted the research community, by being reviewed, accepted and then actively developed and further enhanced into the Attic software. This is a powerful indicator of the usefulness and need for ADICS, and can be seen as a positive qualitative assessment the ADICS ideas within the broader scientific community.

Another important factor in determining the applicability of ADICS and its Attic implementation is quantitative evaluation. Following the simulations of the ADICS architecture (see §4.5), a prototype implementation was developed and showcased at EDGeS's final review. During the EDGI project, this software was further enhanced into the Attic implementation (see Chapter 5) , which is REST-based and uses standard transport protocols (e.g., HTTP/s). The Attic software was first tested at the computing lab at Cardiff University, and later deployed to the EDGI infrastructure. In this section, the results of the Attic testing are given, which compare it to BOINC and also demonstrate how the network functions.

6.2.1 Laboratory Testing

For the laboratory tests [130], which occurred in 2011 and were led by Abdelhamid Elwaer, 19 Linux machines from the computing lab at the School of Computer Science and Informatics were used. Of these 19 machines, 18 ran various combinations of clients and Data Centers. Each of these machines was a 2.8 GHz Pentium 4 with 2 GB of RAM. The 19th machine was a 2.0 GHz Dual Core Pentium with 3 GB RAM and was used to run either the Data Lookup Server or the BOINC data server, depending on the experiment.

The machines were connected through a high-speed LAN network, where the port speed could be throttled to simulate lower throughput. Thus a subset of the machines was set to 10 Mbp/s, while others were configured for 100 Mbp/s. This enabled a simulation of "home users," connected via broadband connections and also higher-speed project servers.

Figure 6.1 shows an evaluation of a single BOINC Web server compared with various Attic network configurations. For each simulation, the same 10 MB data file was requested, but the number of clients varied among one, three, and nine. As the figure indicates, Attic gives similar results when running a single Data Cen-

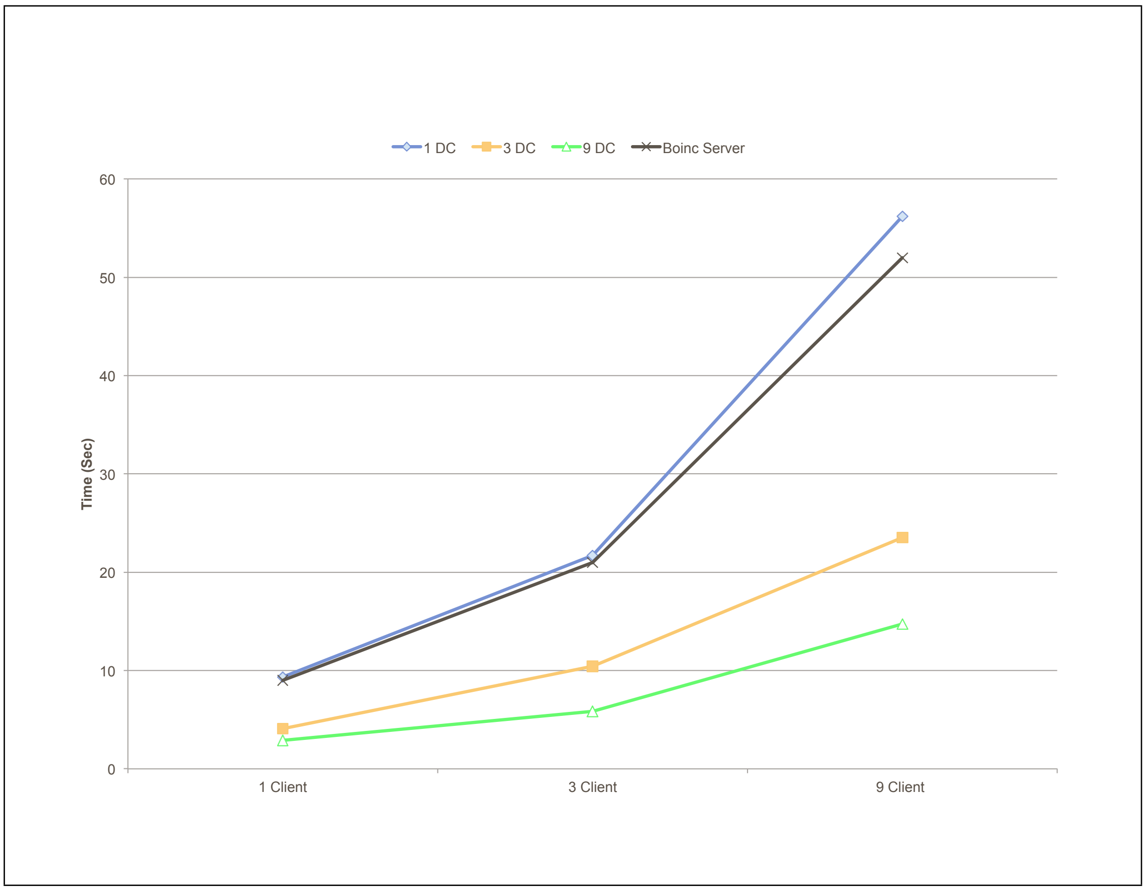


Figure 6.1: Gives testing results showing how an increase in the number of Data Centers and servers helps the network scale as demand increases.

ter, with slightly reduced network speeds. This is due to the message overhead of negotiating an Attic transfer (i.e., querying for individual chunks, reassembling, and prioritizing endpoints) versus a single HTTP request. However, as Attic adds additional Data Centers, the download time decreases and the benefits of the network become apparent. This is a similar effect to adding additional BOINC mirrors; however, the cost of adding an additional Attic Data Center is much lower, since the entire data set (which can be hundreds of gigabytes) does not need to be mirrored.

These tests show results very similar to the results of the simulations described in §4.5. This is very promising for Attic (and subsequently ADICS) because achieving such results was the ultimate goal set forth when designing the ADICS network and undertaking this research project.

6.2.2 Production Testing

The Attic software has been deployed to most of the partner institutions within the EDGI project as part of its Service Activity (SA1). Table 6.1 shows the partner institutes in EDGI, as well as their locations. Each has been given resources to deploy the Attic software. As can be seen from the table, the potential Attic network spans a wide geographic area throughout Europe. This makes the EDGI deployment an ideal test case for Attic deployment, as it represents a widespread virtual organization, with differing network resources and proximity.

Table 6.1: EDGI Attic Infrastructure – Partners

Institute	**Abbreviation**	**Country**
Laboratory of Parallel and Distributed Systems	MTA SZTAKI	Hungary
University of Westminster	UoW	United Kingdom
University of Paderborn	UPB	Germany
University of Copenhagen	UCPH	Denmark
AlmereGrid	—	The Netherlands
University of Coimbra	FCTUC	Portugal
University of Zaragoza	UNIZAR	Spain
Cardiff University	CU	United Kingdom
National Center for Scientific Research	CNRS	France
National Institute for Research in Computer Science and Control	INRIA	France

Within the EDGI infrastructure, each partner's Attic deployment assumes one or more roles (e.g., Data Center and Data Lookup Service). Table 6.2 shows the distribution of Attic entities within EDGI and their hostnames. Due to the concentration of Attic-centric development at Cardiff University, it hosts the central Data Lookup Service, as well as the main Data Seed. Other EDGI institutions are delegated Data Centers, which (in an ideal world) would spread the Attic network load evenly across the project partner sites.

Table 6.3 gives the details of each Attic instance that is currently deployed. All Attic-hosting machines are Linux-based, although with differing distributions (e.g., Centos, Debian, Ubuntu, and Scientific Linux). The decision to use Linux was not due to a fundamental requirement of Attic, as it is written in Java and could also easily run on Windows or OSX. Rather, the choice was made in part because of the powerful security and configuration mechanisms of Linux, which facilitate

Table 6.2: EDGI Attic Infrastructure – Role Distribution

Role	**Institute**	**Host**
Data Center	MTA SZTAKI	attic.lpds.sztaki.hu
Data Center	UoW	attic.cpc.wmin.ac.uk
Data Center	UPB	attic.edgi.pc2.uni-paderborn.de
Data Center	UCPH	edgi-attic.nbi.dk
Data Center	AlmereGrid	server23.almeregrid.nl
Data Center	FCTUC	edgi.dei.uc.pt
Data Center	UNIZAR	attic.ibercivis.es
Data Lookup Service	CU	voldemort.cs.cf.ac.uk
Data Seed	CU	s-vmg.cs.cf.ac.uk
Data Center	CU	s-vmh.cs.cf.ac.uk
Data Center	CNRS	— *
Data Center	INRIA	— *

* At the time of this writing, the Attic software is in the process of being deployed at this site.

easy deployment and configuration, and also due to the individual preferences of project partners.

Table 6.3: EDGI Attic Infrastructure – Details

Host	**HDD**	**Memory**	**Architecture**	**Cores**
attic.lpds.sztaki.hu	500GB	2GB	AMD VM	1
attic.cpc.wmin.ac.uk	1TB	4GB	AMD Opteron	4
attic.edgi.pc2.uni-paderborn.de	300GB	8GB	Intel Xeon	8
voldemort.cs.cf.ac.uk	1TB	6GB	AMD Opteron	4
edgi-attic.nbi.dk	1.1TB	12GB	Intel Xeon	8
server23.almeregrid.nl	1TB	8GB	Intel Xeon	4
edgi.dei.uc.pt	2TB	12GB	Intel i8	8
attic.ibercivis.es	500GB	4GB	AMD	16
s-vmg.cs.cf.ac.uk	75GB	8GB	VM	2
s-vmh.cs.cf.ac.uk	75GB	8GB	VM	1

6.2.2.1 EDGI Deployment Testing

To test Attic, two BOINC projects were setup. The first hosted the Attic Proxy (see §5.3), which was modified to store download metrics on client machines. The second project, entitled “md5hash,” was a new application created specifically for this testing. Md5hash did not process any scientific data, rather it supported a parameter, `cpu_time`, that specified the amount of gigaflops on the client machine it should consume. This enabled precise control over how long each work unit would take to complete its CPU-intensive tasks, letting the testing focus on data transfer. In addition, at the end of its computation md5hash was programmed to transfer the download metrics stored by the Attic Proxy to the BOINC server as its program output. This allowed for the mapping of download times and transfer metrics to individual work units.

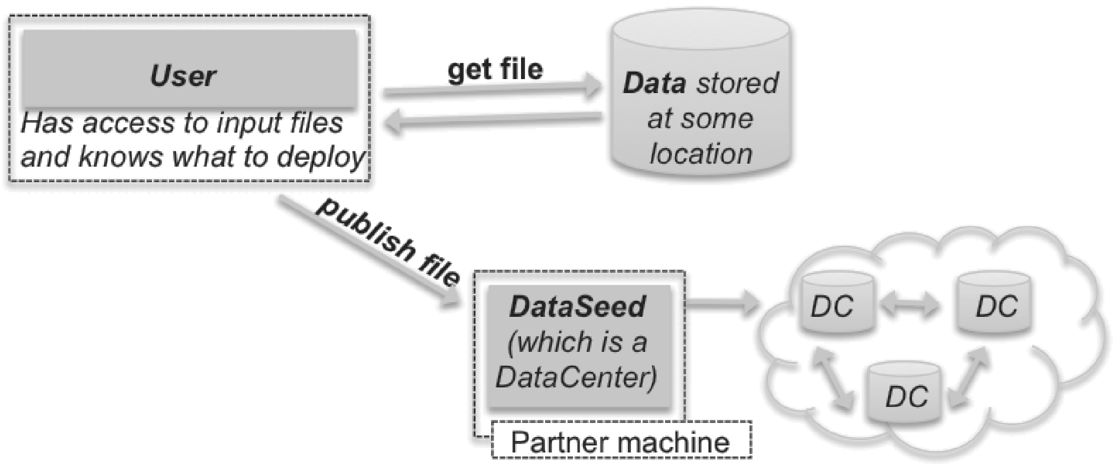

User
Has access to input files and knows what to deploy
get file
Data stored at some location
publish file
DataSeed (which is a DataCenter)
Partner machine
DC
DC
DC

Figure 6.2: Shows how files are deployed by a user and propagated through EDGI's Attic infrastructure.

Three separate tests were performed, using 1 MB, 10 MB, and 100 MB input files. For each run, 100 md5hash work units were created. Client nodes consisted of 25 virtual machines running on an OpenStack Cloud at Cardiff University. Each VM was configured as a 64-bit Linux client with a single core, 15 GB of HDD space, and 2 GB of RAM. The `cpu_time` parameter was set to *one second*, to increase Attic network requests by several orders of magnitude over what 100 clients would normally request. This was done to simulate a higher-volume network such as that of Einstein@Home, where a job takes an average of five hours to run (i.e., 18,000 seconds), and requires only a few (i.e., 6–7) megabytes of input data.

Figure 6.2 shows how files were distributed through the testing. The workflow is the same for the testing described here as it would be for any user of gLite, ARC, or UNICORE that is transitioning jobs from Service to Desktop Grids using the tools described in §5.2.

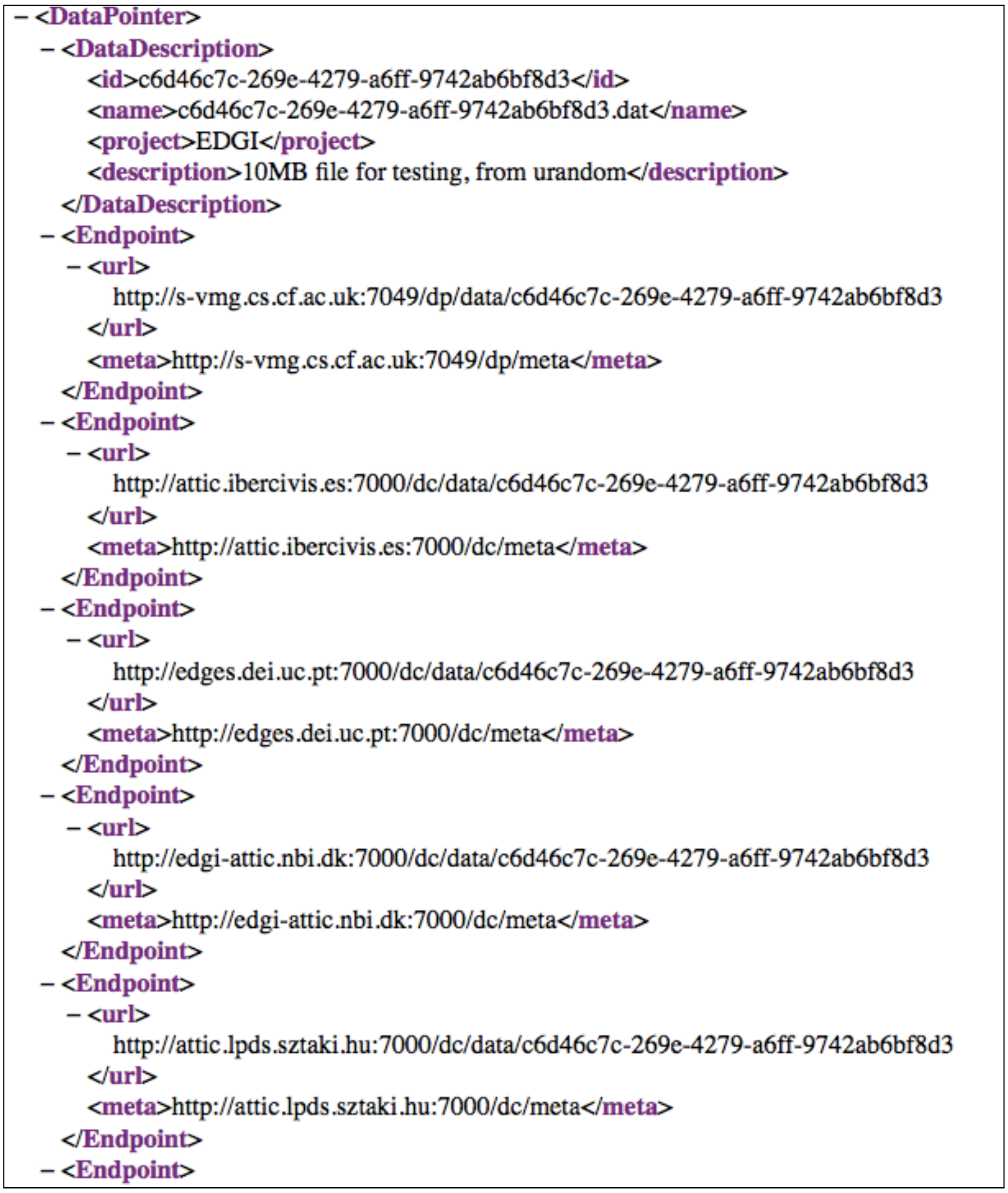
So as not to skew the network results, the Data Center at Cardiff University (see Table 6.3) was disabled during the runs, leaving a total pool of seven Data Centers. If this had not been done, many clients would have requested input files from that machine over the high-speed LAN, and not made use of the other Attic servers. The Data Seed at Cardiff was used in the testing; however, this does not affect where clients download from, due to the automatic de-prioritization mechanisms described in §5.1.5. Figures 6.3 and 6.4 show the relevant metadata for the 10 MB file, such as different endpoints (i.e., Data Centers) on the network that cached the file, and also the individual segment sizes and hash values.

Table 6.4: Attic Deployment Tests – 10 MB

Average Pre Processing Time	0.02s
Average Post Processing Time	0.06s
Duplicate Requests	55
Total # of Requests	2,082
Total Data Centers Utilized	5
Average Data Centers Utilized	2.59
Average Download Time	48.66s
Average Requests per Data Center	416.4
Average Download Time per Chunk	0.19s
Highest Average Mb/s	10.38
Lowest Average Mb/s	0.04
Network Speed	5.46 Mb/s

Tables 6.4 and 6.5 show the results of the testing for the 10 MB and 100 MB files. As can be seen from the tables, the pre- and post- processing time needed by Attic is very minimal when compared to overall download times. This is consistent with the results shown in Figure 6.1, where Attic was compared with BOINC. Interesting to note is the great discrepancy between the fastest and slowest Data Centers in the network for the 10 MB tests, where a total of five Data Centers were used. The fastest was performing at a respectable 10 megabits per second, yet the slowest was barely functioning at all. In the 100 MB tests, only three Data Centers were selected, resulting in an increased overall “Network Speed” due to the removal of the non-performers. On average, however, most Data Centers performed adequately.

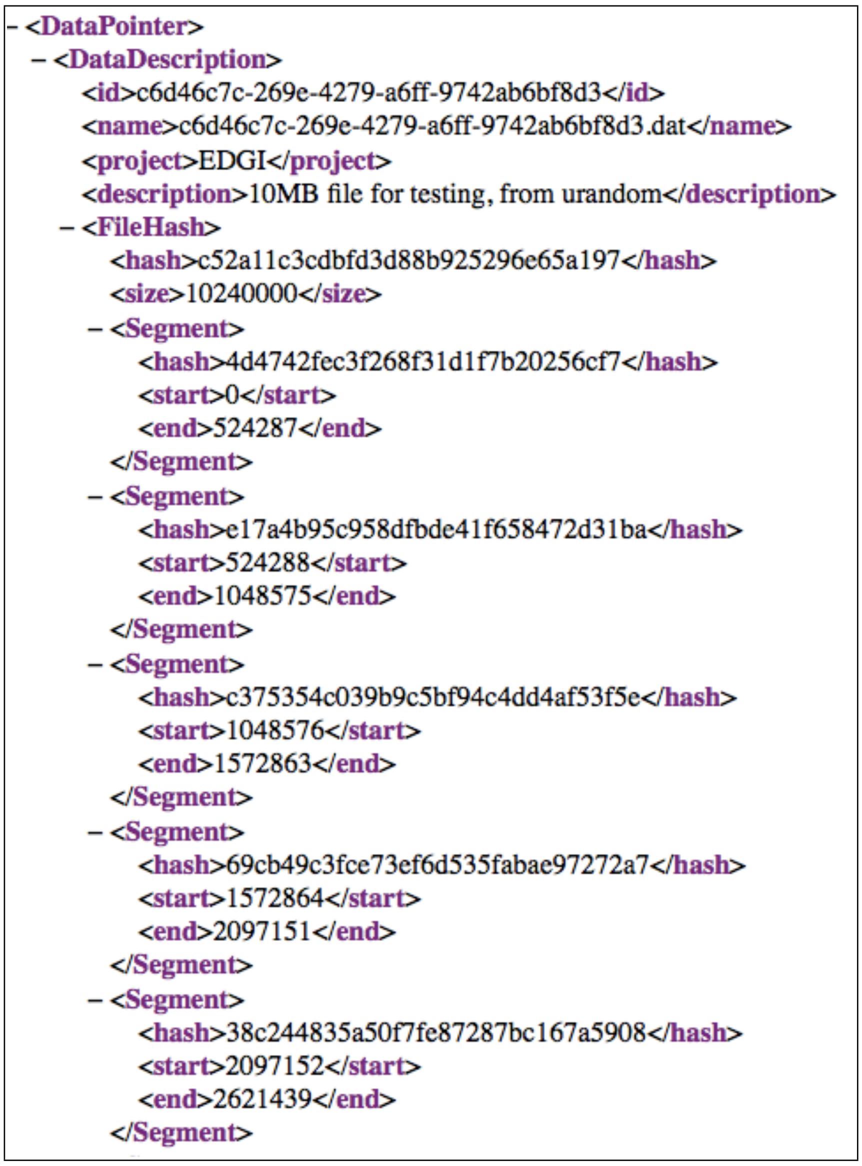
Overall, the testing has shown that the Attic network functions as expected, providing load distribution among the participants. Section 6.3 discusses further work that could take place to enhance the Attic implementation. Particularly rel-



```
– <DataPointer>
  – <DataDescription>
      <id>c6d46c7c-269e-4279-a6ff-9742ab6bf8d3</id>
      <name>c6d46c7c-269e-4279-a6ff-9742ab6bf8d3.dat</name>
      <project>EDGI</project>
      <description>10MB file for testing, from urandom</description>
    </DataDescription>
  – <Endpoint>
    – <url>
        http://s-vmg.cs.cf.ac.uk:7049/dp/data/c6d46c7c-269e-4279-a6ff-9742ab6bf8d3
      </url>
      <meta>http://s-vmg.cs.cf.ac.uk:7049/dp/meta</meta>
    </Endpoint>
  – <Endpoint>
    – <url>
        http://attic.ibercivis.es:7000/dc/data/c6d46c7c-269e-4279-a6ff-9742ab6bf8d3
      </url>
      <meta>http://attic.ibercivis.es:7000/dc/meta</meta>
    </Endpoint>
  – <Endpoint>
    – <url>
        http://edges.dei.uc.pt:7000/dc/data/c6d46c7c-269e-4279-a6ff-9742ab6bf8d3
      </url>
      <meta>http://edges.dei.uc.pt:7000/dc/meta</meta>
    </Endpoint>
  – <Endpoint>
    – <url>
        http://edgi-attic.nbi.dk:7000/dc/data/c6d46c7c-269e-4279-a6ff-9742ab6bf8d3
      </url>
      <meta>http://edgi-attic.nbi.dk:7000/dc/meta</meta>
    </Endpoint>
  – <Endpoint>
    – <url>
        http://attic.lpds.sztaki.hu:7000/dc/data/c6d46c7c-269e-4279-a6ff-9742ab6bf8d3
      </url>
      <meta>http://attic.lpds.sztaki.hu:7000/dc/meta</meta>
    </Endpoint>
  – <Endpoint>
```

Figure 6.3: Shows the portion of a Data Pointer used in the Attic deployment tests that contains the “endpoint” information (i.e., Data Centers) about where files can be retrieved from.

evant to the testing performed here would be the addition of quality of service enhancements for Data Center selection, which would limit the effect of slow or non-performing hosts.



```
- <DataPointer>
  - <DataDescription>
      <id>c6d46c7c-269e-4279-a6ff-9742ab6bf8d3</id>
      <name>c6d46c7c-269e-4279-a6ff-9742ab6bf8d3.dat</name>
      <project>EDGI</project>
      <description>10MB file for testing, from urandom</description>
    - <FileHash>
        <hash>c52a11c3cdbfd3d88b925296e65a197</hash>
        <size>10240000</size>
      - <Segment>
          <hash>4d4742fec3f268f31d1f7b20256cf7</hash>
          <start>0</start>
          <end>524287</end>
        </Segment>
      - <Segment>
          <hash>e17a4b95c958dfbde41f658472d31ba</hash>
          <start>524288</start>
          <end>1048575</end>
        </Segment>
      - <Segment>
          <hash>c375354c039b9c5bf94c4dd4af53f5e</hash>
          <start>1048576</start>
          <end>1572863</end>
        </Segment>
      - <Segment>
          <hash>69cb49c3fce73ef6d535fabae97272a7</hash>
          <start>1572864</start>
          <end>2097151</end>
        </Segment>
      - <Segment>
          <hash>38c244835a50f7fe87287bc167a5908</hash>
          <start>2097152</start>
          <end>2621439</end>
        </Segment>
```

Figure 6.4: Shows the portion of a Data Pointer used in the Attic deployment tests that contains the “chunk” information about files.

6.3 Further Work

The ADICS architecture has remained stable throughout the last two years, during the EDGI project. The concepts of the Data Lookup Service, Data Centers, and Data Seeds, and their entity interactions, have proved useful for the Desktop Grid community and led to the development of Attic. The Attic software is a concrete implementation of the ADICS network design, and thus evolves more quickly and is more subject to feature enhancements and changes in user requirements that do not affect the fundamental underlying network topology of ADICS. It is a rare

Table 6.5: Attic Deployment Tests – 100 MB

Average Pre Processing Time	0.13s
Average Post Processing Time	0.51s
Duplicate Requests	0
Total # of Requests	19,405
Total Data Centers Utilized	3
Average Data Centers Utilized	2.35
Average Download Time	43s
Average Requests per Data Center	6,468.33
Average Download Time per Chunk	0.76s
Highest Average Mb/s	10.75
Lowest Average Mb/s	1.12
Network Speed	6.87 Mb/s

(probably nonexistent) software product that is able to sustain itself without being periodically augmented with new features and updated as the computing environment changes. It is with this in mind that this section seeks to outline ways in which either the ADICS architecture or the Attic software could be improved or enhanced with further work that is beyond the scope of this dissertation.

6.3.1 Distributed Lookup Service

Currently, the metadata tracking entity of ADICS is a single Data Lookup Service. This solution works well for projects that have centralized entities that can manage the DLS and assure its stability, such as BOINC projects or the EDGI project. By keeping metadata management centralized (a concept also used by BitTorrent's trackers), control can be exerted over security mechanisms and data propagation policies without great complexity. However, to leverage the ADICS architecture in a more loosely coupled manner, as would be needed to make it useful for other file-sharing applications, the DLS would benefit from decentralization.

Decentralized metadata management could be accomplished by applying Distributed Hash Table (DHT) concepts (see §2.5.5). DHT enhancements would split responsibility for metadata management among network participants, provide greater scalability, and eliminate the potential bottleneck of a centralized server. There exist proven and available DHT implementations that could be used for this task. Alternatively, metadata management could be delegated to another infrastructure that already implements a DHT. In this scenario, the Data Lookup Service would not be used, and Attic (as an implementation of ADICS) could function as a low-level transport protocol in a framework like BitDew [131], which provides a scalable metadata management layer that would sit on top of Attic in a functional hierarchy.

6.3.2 Monitoring

One very important aspect to having a data distribution system is being able to monitor its performance. Monitoring is not only a way to garner useful results about the data being transferred on the network, but can also be leveraged to provide extensive QoS metrics and aid in server selection and the choice to either expand or contract the network. Since the ADICS network and the subsequent Attic implementation decentralize control of the data distribution, these metrics are not available in any centralized location. The Data Lookup Service knows which Data Centers have a given file, and could potentially keep track of the number of requests made for a given data entry. However, the DLS has no information about which Data Centers ultimately serve a file, or the speeds at which they do so.

Aggregating the individual transfer and storage metrics from each Data Center to a centralized location, such as the EDGI Monitoring system [132], would prove extremely beneficial for overseeing the Attic network and providing future QoS upgrades. Listing 6.1 shows an example of how project-based metrics could be pushed to the monitoring system. In this example, file metrics are combined based upon their project tags, and then uploaded to the monitoring server. This provides a base level of management, and would enable administrators to see how their network is being used and partitioned over the different projects that are making use of it. Listing 6.2 provides a more granular yet verbose view of data transfers. In that example, the overall transfer for individual files is provided, leaving project aggregation for the monitoring subsystem.

Listing 6.1: Project-Based Monitoring Schema

```
<report time="1302361914248" zone="GMT" version="1.1">
    <monitoring_data>
        <report_start> 10-3-2011 10:58:26 </report_start>
        <report_end> 10-4-2011 10:58:26 </report_end>
        <datacenter_ip> 192.168.0.1 </datacenter_ip>
        <upload_aggregate project="edgi_dsp">
           <uploaded unit="KB">45</uploaded>
        </upload_aggregate>
             .
             .
             .
    </monitoring_data>
</report>
```

Listing 6.2: File-Based Monitoring Schema

```
<report time="1302361914248" zone="GMT" version="1.1">
    <monitoring_data>
        <report_start> 10-3-2011 10:58:26 </report_start>
        <report_end> 10-4-2011 10:58:26 </report_end>
        <datacenter_ip> 192.168.0.1 </datacenter_ip>
        <file ID="987654.dat" name="DSPInput.dat"
              project="edgi_dsp">
            <chunk_size unit="KB">10<chunk_size>
            <chunk_requests>840</chunk_requests>
            <total_transferred unit="KB">8400</size>
        </file>
             .
             .
             .
        <xfer unit="MB" direction="UP">500</xfer>
        <xfer unit="MB" direction="DOWN">100</xfer>
    </monitoring_data>
</report>
```

6.3.3 Quality of Service Enhancements

Following the collection of monitoring metrics, greater levels of Quality of Service (QoS) could be applied to the Attic network. As discussed in §5.1.5, the current choice of download endpoints (i.e., Data Centers) is done through a basic Round Trip Time (RTT) query. However, a snapshot of network latency, which is what a RTT gives, can be a poor metric for choosing download locations. The QoS work outlined in Appendix B shows promise in creating a robust selection criterion for Data Centers, basing selection on historical performance metrics and trust values. If that system were combined with the verbose monitoring information collection proposed in §6.3.2, it would make a powerful addition to Attic, increasing network speeds and reliability.

Another important QoS metric is ensuring that the system has sufficient space and bandwidth to store and serve the requests that are being made. By design, each Data Center determines its own bandwidth and storage contributions. This is an important element to ensure participation at all levels, rather than enforcing strict quotas. However, as these metrics are not centrally tracked, the main metadata management server for the network (i.e., the DLS) cannot determine if it should reject further caching requests to ensure the delivery and storage quality of existing artifacts. Rather, it is expected that Data Centers will remove stale data to free-up capacity, and/or the network will scale by increasing the size of the Data Center overlay. This scenario allows for the greatest freedom and autonomy for the Data Centers, yet creates a potential quality of service deficit, and should therefore be explored further.

6.3.4 General Improvements

There are several additional areas in which the general performance and stability of the Attic implementation could be improved:

- The middleware used to implement wire-level transport and XML parsing in Attic [133] should be changed to one that is more current and updated. There have recently been reports of network errors causing Data Centers to throw exceptions when they receive invalid messages (e.g., XML and JSON), rather than gracefully recovering. It is believed these stem from inconsistencies in the XML parser libraries in the EDGI production infrastructure. Updating the underlying message-layer code that serializes/deserializes the XML and JSON messages should be able to alleviate this problem. Jersey [134] is an excellent candidate, as it is a solid and up-to-date software package that supports multiple mime types for both input and output.

- The current mechanism for transferring data from a Service Grid to a Desktop Grid is to use a Data Seed to stage the data to the network. Currently, there is no round-robin or other load-balancing mechanism in place to distribute loads among multiple Data Seeds. In a production environment, this can lead to centralized points of failure.
- The Attic implementation for storing metadata on the Data Lookup Servers and Data Centers is rudimentary, using in-memory objects that get periodically persisted to flat files. This should be upgraded to use a RDMS, which will increase reliability and help aid recovery in the event of sudden machine failure/reboots.
- Integrating with existing systems and uptake is important for Attic/ADICS to succeed. Further efforts should be put into finalizing the integration with BOINC and ultimately using Attic to run data-intensive applications on a production level.

CHAPTER 7

Conclusions

Software evolves and computing landscapes change. Since the onset of this thesis, much focus in the research and commercial communities has turned from Grid to Cloud computing infrastructures, and all the while, the data needs of the Desktop Grid community have continued to grow. In each of these environments, data management, with all that it entails, remains a topic of great interest, investment, and research.

The hypothesis supporting the research undertaken here is that peer-to-peer file sharing strategies can successfully be applied to the volunteer computing application domain. Unlike Service Grids and Clouds, which are generally centrally managed with large support infrastructures, most Desktop Grids are comprised of volatile, heterogeneous, and geographically distributed volunteer computing resources. These characteristics, combined with the large scale, legacy applications, and diverse user communities inherent in volunteer Desktop Grids make them an ideal platform in which to propose, and test, new data management paradigms.

After a thorough investigation of the proposed hypothesis, that "peer-to-peer file sharing can be successfully applied to the volunteer computing application domain," this dissertation asserts it is true. This contention is validated both by the work presented here and also through the following metrics: first, the ADICS ideas were accepted (and significantly funded) in two fair sized European Union projects; second, the research community acknowledged ADICS as valid research through peer-reviewed publications; and finally, Desktop Grid community acceptance was achieved and extensive progress was made towards integrating Attic into existing middleware systems.

7.1 Summary

The advantage and appeal of volunteer computing is its cost effectiveness, giving access to potentially tens of thousands of resources for a small fraction of the "real" costs.[1] Given the relatively low budgets of volunteer computing projects (the vast majority are scientific research) and their reliance on donated resources, it is unlikely that the future data-distribution paths for these projects will be to simply offload their storage and network needs to commercial Clouds. A distributed load-balancing solution that leverages the (existing) network and storage capabilities of donor machines would be an ideal (and low-cost) solution.

The need for a new data-distribution paradigm is a known problem within the community, not only as a way to enable new applications to make use of volunteer networks, but also to facilitate current trends toward using virtual machines to provide a homogeneous, configurable, and safe execution environment. In all scenarios, data loads are increasing, often by orders of magnitude, and the urgency of solving the "data problem" is becoming more pronounced.

Many ideas from peer-to-peer networking can be successfully applied to the Desktop Grid computing domain. Important to the process of applying P2P technologies to Desktop Grids is understanding the community context and identifying their requirements. This is especially important when considering a "donation-based" volunteer computing environment, where donor retention is of paramount importance. An out-of-the-box P2P software infrastructure, such as BitTorrent, is unable to address many of the issues present in Desktop Grids, especially as they relate to participation and user security. Likewise, generic P2P middleware projects, such as JXTA, often impose network topologies and message relaying infrastructures that can be an obtuse match for volunteer networks, and that would require extensive modifications to function properly.

[1] In volunteer computing, costs are shifted from centralized projects to the donors. It is the donors who manage the computer resources, providing periodic hardware upgrades and paying maintenance costs such as electricity and network connectivity charges.

In addition to the data needs within Desktop Grids, there has recently been a push to provide migration mechanisms to move jobs from Service to Desktop Grids. This is partially a sustainability issue and partially a practical resource-utilization one. For sustainability, the costs of maintaining and building new Service Grids is immense, and any solution that can minimize Service Grid resource requirements is welcomed. As a practical matter, there are a large number of "embarrassingly parallel" applications that run on expensive, tightly coupled Service Grid infrastructures that could also function in a Desktop Grid environment. Often the barriers to transitioning these jobs to Desktop Grids are the high upfront cost of configuring Desktop Grid middleware and the issue of attracting donors. The EDGeS and EDGI projects have provided bridging mechanisms to ease this transition; however, they each require a robust (preferably decentralized) data management solution. This is due to the nature of Service Grid data that are often in secured storage and need to be publicly exposed, and also to the nature of Desktop Grids, which require input files to be distributed to each Worker node over a Wide Area Network (WAN), a task that necessitates large network resources.

The Peer-to-Peer Architecture for Data-Intensive Cycle Sharing (ADICS) and its subsequent REST-based implementation, Attic, seek to fill these voids. ADICS takes many of the best practices and techniques from the P2P world and fuses them with the requirements of Desktop Grids (especially volunteer computing communities) to provide a new approach for data distribution. The architecture proposed by ADICS distributes load by providing a network with a partial replica overlay, as opposed to the current practice of full-mirroring to scale. The idea proposed in ADICS of partial replicas, combined with dynamic network strategies, lowers the barriers to participation and allows for network (re)configuration and load balancing. Yet unlike many P2P networks, vital control remains in the hands of project administrators and data-sharing is not forced on all network entities.

For Service to Desktop Grid migration, ADICS provides a system where data can be *pushed* to the P2P environment by the Service Grid nodes that have the necessary security credentials to access the data. This not only alleviates the security problems associated with somehow allowing untrusted users access to Service Grid storage elements, but also offloads the data distribution, making the integration of Service and Desktop Grids more accessible.

7.2 Future Work

Section 6.3 identified several key improvements and further work that could be undertaken to improve the ADICS architecture and its Attic software implementation. It should be noted, however, that beyond simply making the software more robust and increasing its impact on the existing Desktop Grid communities, there are a number of interesting new research areas that could benefit from ADICS.

Here I propose areas of future work where I think it would be interesting to continue ADICS research. ADICS was designed to be a generic data sharing architecture, albeit one that could be used successfully to distribute Desktop Grid data. Likewise, the core Attic implementation is not tied to a particular problem domain or application context. It is implemented using Service Oriented Architecture (SOA) and Representational State Transfer (REST) principles. As such, the possibilities and range for using ADICS (and Attic) for further data management investigation are broad.

One area in which I would be especially keen to transition ADICS into is the mobile application domain. I believe peer-to-peer techniques, combined with encryption, error correction, and erasure coding, can be effective tools for data retention and sharing in highly volatile networks. In such an environment ADICS would be able not only to aggregate storage capacity, but also could be adapted to share latent bandwidth. This could be useful for tasks ranging from combining upstream bandwidths to enable high-definition video streaming (e.g., from a kids' soccer game) to low-cost roaming on ad hoc data networks.

Another intriguing topic is leveraging peer networks to create distributed shared disks – that would live beyond any particular computing device or user, similar to Dropbox [135] but, secure, encrypted, and not centrally managed by a corporate entity. It is not unrealistic to have data living in a "peer cloud," where cost can be minimized while concurrently protecting privacy and intellectual property rights. Access Control Lists (ACLs) could be applied to the data to facilitate easy sharing and management. The P2P filesystem could even be integrated into client operating systems, by using technologies such as FUSE [136] – making it simple to drag and drop files into a "distributed and shared disk." With this technology freely available, secured, and private, it could provide a very useful tool for backing up and sharing personal information and files. Personally, I would be very interested in using such a system as it would greatly simplify coordinating family photo albums, video libraries, and data backups.

APPENDIX A

Code Examples

This appendix gives several typical examples of using Attic and instantiating many of its network roles. For further instructions on how to use Attic, as well as JavaDocs and other auxiliary information, refer to the project website under:

`http://www.atticfs.org`

Note that the Attic software is available under a liberal Apache license, making it available for public, private, and commercial use. The latest source code for Attic and the LibAFS proxy application can be found on GitHub under the following:

`https://github.com/ikelley/Attic`

`https://github.com/keyz182/afs_proxy`

Attic URL handler

The Attic protocol handler provides a way to read `attic://` URLs directly into Java programs, treating them as ordinary (i.e., built-in, such as *http* and *ftp*) URLs. This allows applications, such as XtremWeb (see §5.4), to reference `attic://` URLs anywhere in their code after the handler has been registered and the appropriate libraries have been included. This is an extremely powerful tool for integration, as it allows Attic to be incorporated into any number of third-party middleware systems with very minimal modification (i.e., only a few lines of code).

Listing A.1 gives an example of how, after registration of the handler, an `attic://` URL can be easily read using a standard `InputStream`.

Listing A.1: Code Example: Attic URL Handler

```
// register the attic: and attics: (secure) URL schemes
// NOTE: this will not destroy existing mappings.
AtticProtocol.registerAttic();
// The URL should point to a DataPointer
URL url = new URL("attic://www.example.org:9876"
                 + "/dl/meta/pointer/1234567890");
File file = new File("out.dat");

// open the stream
InputStream in = url.openStream();
FileOutputStream fout = new FileOutputStream(file);

byte[] bytes = new byte[8192];
int c;
while ((c = in.read(bytes)) != -1) {
   fout.write(bytes, 0, c);
}
fout.flush();
fout.close();
in.close();
```

Creating a DataWorker and Downloading Data

The `DataWorker` role showcases how to download data from the Attic network. The mechanisms and workflow for retreiving data from the network are the same, regardless of the client, be it a `DataWorker`, `DataCenter`, or other network entity. Listing A.2 dhows how an Attic configuration is initialized, and then the `DataPointer` is retrieved, with a callback registered for when the final data arrive back at the requestor. In the default configuration, the resulting data will be written to file in the *dw* directory.

Listing A.2: Code Example: DataWorker Instance Downloading a File

```
tic = new Attic();
attic.setRole(StringConstants.ROLE_DW);

ConfigServiceRole confService = new ConfigServiceRole();
attic.attach("attic", confService);

DataWorker dw = new DataWorker();
attic.attach(StringConstants.DATA_WORKER, dw);
attic.init();

try {
   dw.getPointer(new Endpoint("https://example.org/"
       + "dl/meta/pointer"));
} catch (IOException e) {
   e.printStackTrace();
}
```

Creating a DataLookup Service

The `DataLookup Service` provides the fundamental role of storing the authoritative metadata information about files and which `DataCenters` have replicated them on the network. For Attic to function properly, a `DataLookup Service` must exist somewhere in the network, as the other network entities use it as the bootstrap endpoint to initiate network queries. Listing A.3 shows how a basic instance of a `DataLookup Service` can be initialized on the network.

Listing A.3: Code Example: DataLookup Service

```
Attic attic = new Attic();
attic.setRole(StringConstants.ROLE_DL);
attic.attach("attic", new ConfigServiceRole());

DataLookup dl = new DataLookup();
attic.attach(StringConstants.DATA_LOOKUP, dl);
attic.init();
```

Creating a DataCenter with a Custom Query Interval

`DataCenters` periodically query the `Scheduler` to request new replication work. The default query interval is one hour; however, this can be customized by system administrators to more adequately match their application's needs. For example, when testing Attic, a much shorter query interval would be recommended, as this would increase the speed of propagation on the network, with the additional cost of increased message queries. For less frequently updated networks, the query interval could be set to daily or weekly, and even synchronized to query for network updates during off-peak hours to minimize network demand. Listing A.4 shows a `DataCenter` being initialized with a query interval of two minutes.

Listing A.4: Code Example: DataCenter with Custom Query Interval

```java
Attic attic = new Attic();
attic.setRole(StringConstants.ROLE_DC);
attic.getDataConfig().setDataQueryInterval(120);
attic.setBootstrapEndpoint("https://example.org"
    + "/dl/meta/pointer");

DataCenter dc = new DataCenter();
attic.attach(StringConstants.DATA_CENTER, dc);
attic.init();
```

Attic Instance Supporting Multiple Roles

In Attic, there are multiple roles that can be performed by network entities, such as a `DataCenter`, `DataSeed`, or `DataLookup Service`. Often these roles are logically split between different virtual or physical machine hardware, to provide load balancing. However, it can also be desired to run multiple roles on the same Java Virtual Machine (JVM) and network port. This might be to accommodate network firewall policies that are allowing only a limited number of ports, or to minimize the overhead of the JVM. It can be foreseen that often the `DataLookup Service` and either a `DataCenter` or `DataSeed` could be combined.

As the Attic implementation references each role in a different namespace (e.g., *dc*, *dl*, or *ds*), there are no conflicts within the software when supporting more than one role within the same Attic instance. Listing A.5 shows a

DataCenter and a DataLookup Service being started in the same instance (and likewise the same port).

Listing A.5: Code Example: Attic Instance Supporting Multiple Roles

```java
Attic attic = new Attic();
attic.addRole(StringConstants.ROLE_DC);
attic.getDataConfig().setDataQueryInterval(120);
attic.setBootstrapEndpoint("https://example.org"
    + "/dl/meta/pointer");

DataCenter dc = new DataCenter();
attic.attach(StringConstants.DATA_CENTER, dc);
attic.addRole(StringConstants.ROLE_DL);

DataLookup dl = new DataLookup();
attic.attach(StringConstants.DATA_LOOKUP, dl);
attic.init();
```

Secured DataSeed

The DataSeed node in Attic enables third parties, such as command-line utilities, to publish files to the Attic network. It is an extremely useful function when integrating Attic with legacy applications, as discussed in §5.2. Listing A.6 shows how to create a DataSeed that is secured (using certificates) and requires client authentication to publish files to the network.

Listing A.6: Code Example: Secured DataSeed

```
Attic attic = new Attic();
attic.setBootstrapEndpoint("https://example.org"
    + "/dl/meta/pointer");
attic.addRole(StringConstants.ROLE_DS);
DataSeed dataSeed = new DataSeed();
attic.attach(StringConstants.DATA_PUBLISHER, dataSeed);

// set up keystores and security config.
// These are stored, so actually this doesn't have to
// be done on every start up
SecurityConfig sc = attic.getSecurityConfig();
sc.setSecure(true);
sc.setRequireClientAuthentication(true);

URL trustUrl = getClass().getClassLoader()
    .getResource("keystores/trust.keystore");
String trust = trustUrl.toString();

Keystore ts = new Keystore(trust,
    "ca-keystore-password", "ca-keystore-alias");
ts.setName(key + "-trust.properties");
sc.addTrustStore(ts);

URL keyUrl = getClass().getClassLoader()
    .getResource("keystores/seed-key.keystore");
String keystore = keyUrl.toString();

Keystore ks = new Keystore(keystore,
    "seed-keystore-password", "seed-keystore-alias");
ks.setName("seed-key.properties");
sc.addKeyStore(ks);

attic.init();

// NOTE: adding DN names after init method
dataSeed.addIdentity("CN=machine1.example.org,
    OU=attic, O=Example, L=Cardiff, ST=Wales, C=UK",
StringConstants.CACHE_KEY);
dataSeed.addIdentity("CN=machine2.example.org,
    OU=attic, O=Example, L=Cardiff, ST=Wales, C=UK",
StringConstants.CACHE_KEY);
```

APPENDIX B

Impact Details

This appendix provides an overview of the projects that the ADICS research has had an impact on. It is useful to recount the trajectory of ADICS development through each of these projects, as it helps to demonstrate how this thesis evolved over time by gaining greater exposure and insight into the problem domains. Beyond its natural evolution through greater exposure, the ADICS architecture was also shaped through input from external user-groups and partners who identified new areas in which the protocol could be applied.

The details of those projects, in rough chronological order, are given in the remainder of this appendix.

Secure Decentralized Data Sharing in Dynamic Distributed Networks

My research in the data management aspects of distributed computing was first undertaken through an Engineering and Physical Sciences Research Council (ESPRC) grant (EP/C006291/1) to investigate: “Secure Decentralized Data Sharing in Dynamic Distributed Networks.” The goals of that project were to identify new mechanisms to distribute Desktop Grid (notably Einstein@Home, see

§2.3.3.2) input data. This was seen as advantageous because, in the case of Einstein@Home, new gravitational wave datasets were coming online, and their mirroring solution for distributing files to workers required many powerful centralized network servers, as well as system administrator oversight. Partitioning the data over a widespread network would reduce centralized loads and allow for greater scaling and higher-resolution input data. The project, with its goals of investigating secure decentralized data sharing mechanisms for dynamic distributed networks, provided much of the backbone research for the development of ADICS and Attic. Ultimately, the ESPRC project was successful, resulting in the development of the ADICS architecture and its first prototype implementation.

During the course of the aforementioned ESPRC project, the ADICS ideas were disseminated through the FP6 CoreGRID Network of Excellence [137], a new P2P research group was formed, and ties were made with colleagues in the Service and Desktop Grid communities. ADICS became an interesting proposition for Desktop Grid data distribution, as well as a way to help transition Service Grid data to a Desktop Grid environment. Discussions about how ADICS could be leveraged in a hybrid Service and Desktop Grid environment led to inclusion of ADICS in a new EU proposal entitled “Enabling Desktop Grids for e-Science” (EDGeS).

Enabling Desktop Grids for e-Science (EDGeS)

The Enabling Desktop Grids for E-Science (EDGeS) [50] was an EU FP7 Infrastructures project that aimed to provide integration paths between Service Grids and Desktop Grids. The targets of the project were user communities that required large computing power that was not available or accessible in available scientific e-Infrastructures. In order to support the specific needs of these scientific and other communities, EDGeS worked to interconnect the large European Service Grid infrastructure (EGEE) with existing Desktop Grid (DG) systems such as BOINC and XtremWeb.

Building a bridge between Service and Desktop Grids provided users with the ability to transparently execute applications on either platform involved in the new integrated infrastructure. Using the advantages of both approaches allowed the EDGeS infrastructure to take a major step towards a European-wide scientific grid, where an extremely large number of resources could be integrated to support grand-challenge scientific and other applications. The involvement of low-cost (volunteer) Desktop Grids into the European scientific grid infrastructure enabled EDGeS to boost the number of available resources and therefore contribute to a sustainable European Grid infrastructure.

Cardiff University was in charge of the “Data Access” joint research activity (JRA 3). JRA3, under my direction, further developed ADICS, focusing both on building a network to share and distribute data *within* Desktop Grids, and also on *migrating* data to the Desktop Grid environment from Service Grid infrastructures. To this end, JRA3 further honed the research started in the ESPRC proposal, while broadening it to incorporate other Desktop Grids and address the issue of data migration.

Transitioning of data between these two systems was a very important element to EDGeS because it required some way to distribute gLite (i.e., Service Grid) data that could adapt to varying input-file sizes and replication factors without unduly stressing or exposing the Service Grid layer. As the EGEE security infrastructure and policies prevented access to local files from foreign and untrusted hosts, there was no obvious way to transfer these files, except to institute a large array of Web-server mirrors, which could prove costly and obtuse to manage. An ADICS solution, which dynamically organized partial replicas throughout the network, was seen as a way to greatly reduce resource requirements while at the same time providing the needed scalability.

EDGeS was a successful project, resulting in much cross-institutional research, interesting results such as the simulations shown in Chapter 4, and the employment of several students and researchers at Cardiff. To culminate the data activities of the project, a prototype implementation of the ADICS system was created and successfully demonstrated at the final EU project review.

Project facts:

Contract number:	RI-211727
Project type:	I3
Start date:	01/01/2008
Duration:	27 months
Total budget:	2,871,480€
Funding from the EC:	2,450,000€
Total funded effort in person-month:	409.6

Project partners:

1*	Magyar Tudomanyos Akademia Szamitastechnikai es Automatizalasi Kutatointezete	MTA SZTAKI	HU
2	Center of Extremadura for Advanced Technologies	CIEMAT	ES
3	Foundation for the Development of Science and Technology in Extremadura	FUNDECYT	ES
4	Institut National de Recherche en Informatique en Automatique	INRIA	FR
5	University of Westminster	UoW	UK
6	Cardiff University	CU	UK
7	Faculty of Sciences and Technology of the University of Coimbra	FCTUC	PT
8	Stichting AlmereGrid	AlmereGrid	NL
9	Institut National de Physique Nucléaire et de Physique des Particules	IN2P3	FR

* Project coordinator.

European Desktop Grid Initiative (EDGI)

The European Desktop Grid Initiative (EDGI) [113] followed the EDGeS project as a way to transition much EDGeS' work into a production infrastructure, while concurrently expanding the infrastructure to include additional Desktop and Service Grids as well as Cloud computing infrastructures. EDGI supports user communities in both the European Grid Initiative (EGI) and National Grid Initiatives

(NGI). The target user groups in EDGI are the heavy users of Distributed Computing Infrastructures (DCIs) that require an extremely large number of CPUs to complete their simulations. To achieve greater scalability and facilitate a more efficient use of resources, EDGI augments existing DCIs (typically composed of clusters and supercomputers), by extending them to include public and institutional Desktop Grids and Clouds. The long-term goals of EDGI are to provide a sustainable European computing infrastructure that includes a robust system of federated Desktop Grids.

Through the strength and success of the ADICS research ideas developed in EDGeS, Cardiff University was again involved in the proposal submission and subsequent (successful) project. With regard to data management and the work presented here, EDGI provided a forum for transitioning ADICS' core research and prototype into a concrete reference implementation (see Attic in Chapter 5), a process that had only begun to be realized inside the EDGeS project.

With the additional support of the EDGI project, Attic was able to be developed. Attic created a transparent and easy-to-integrate data management system that supported the legacy software systems being used in production Service and Desktop Grids. Integration was done primary through the use of Representational State Transfer (REST) principles, using standard HTTP/S as the underlying transport mechanism. This enabled the Attic network to be reachable from the additional ARC (see §5.2.2) and UNICORE (see §5.2.3) Service Grid infrastructures that were added as target communities in EDGI. The integration enhancements and implementation improvements also made Attic a useful candidate for BOINC and XtremWeb data distribution.

Project facts:

Contract number:	RI-261556
Project type:	CP-CSA
Start date:	01/06/2010
Duration:	27 months
Total budget:	2,436,000€
Funding from the EC:	2,150,000€
Total funded effort in person-month:	281

Project partners:

1*	Magyar Tudomanyos Akademia Szamitastechnikai es Automatizalasi Kutatointezete	MTA SZTAKI	HU
2	Institut National de Recherche en Informatique en Automatique	INRIA	FR
3	Centre National de la Recherche Scientifique	CNRS	FR
4	University of Westminster	UoW	UK
5	Cardiff University	CU	UK
6	University of Zaragoza	Unizar-Ibercivis	ES
7	Faculty of Sciences and Technology of the University of Coimbra	FCTUC	PT
8	Stichting AlmereGrid	AlmereGrid	NL
9	University of Paderborn	UPB	DE
10	University of Copenhagen	UCPH	DK

* Project coordinator.

Distributed Audio Retrieval Using Triana (DART)

In addition to the projects mentioned above, Attic (and therefore ADICS) has had an impact on local research within the Distributed Computing Group at Cardiff University. The Distributed Audio Retrieval using Triana (DART) [138, 139] project is an example of this. DART worked to build a decentralized overlay for the processing of audio information for applications interested in Music Information Retrieval (MIR). The project was partially funded under the data research activities of both EDGeS and EDGI to prototype job transitions between Service and Desktop Grids. This resulted in DART implementations for both XtremWeb and BOINC, as well as plans to integrate the data retrieval mechanisms of DART (which were centralized) with the Attic network. Attic integration, however, was not fully realized before the completion of the DART project.

Optimisation Techniques for Data Distribution in Volunteer Computing

The Attic network was also used as the base software for a thesis on Quality of Service (QoS) in distributed networks, entitled "Optimisation Techniques for Data Distribution in Volunteer Computing." In this work, partially funded by the EDGI project, Cardiff Ph.D. student Abdelhamid Elwaer enhanced the selection mechanisms of Attic to use availability, honesty, and speed heuristics to determine Data Center selection. As can be seen in the experiments presented in Elwaer's work [130, 140, 141], where a 10 MB file was downloaded by a varying number of clients and Data Centers, the addition of advanced QoS metrics greatly enhances the speed and reliability of the Attic network. The use of Attic outside of the projects it was first involved in is significant because it shows that the work presented here is robust enough to be used as the basis for further research.

APPENDIX C

Publication List

The following lists the publications (and posters) most relevant to this dissertation. I either wrote or had a significant contribution to each of these journal articles, conference and workshop papers, book chapters, or technical reports. Many of them formed the basis for the research presented here. The list is presented in reverse chronological order; it should be noted that many papers originated as technical reports, and were then extended into full conference papers or journal articles. Likewise, several workshop papers were selected for inclusion in collections or books.

I. Kelley, T. Kiss, and I. Taylor “Secure Decentralised Data Sharing in Dynamic Distributed Networks.” Poster at the RCUK Cybersecurity Research Showcase. London, UK. 23 November 2011.

I. Kelley and I. Taylor, “A Peer-to-Peer Architecture for Data-Intensive Cycle Sharing,” in *Proceedings of the First International Workshop on Network-Aware Data Management (NDM 11)*. Seattle, WA, U.S.A. 14 November 2011.

A. Elwaer, A. Harrison, I. Kelley, and I. Taylor. “Attic: A Case Study for Distributing Data in BOINC Projects,” in *Proceedings of the Fifth Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2011)*. Anchorage, AL, U.S.A. 20 May 2011.

T. Kiss, I. Kelley, and P. Kacsuk, “Porting Computation and Data Intensive Applications to Distributed Computing Infrastructures Incorporating Desktop Grids,” in *Proceedings of the International Symposium on Grids and Clouds (ISGC)*. 25 March 2011.

C. Mastroianni, P. Cozza, D. Talia, I. Kelley, and I. Taylor. “A Scalable Super-Peer Approach for Public Scientific Computation.” Journal of Future Generation Computer Systems. Volume 25 , Issue 3, pp. 213–223, March 2009.

Z. Balaton, Z. Farkas, G. Gombas, P. Kacsuk, R. Lovas, A. C. Marosi, A. Emmen, G. Terstyanszky, T. Kiss, I. Kelley, I. Taylor, O. Lodygensky, M. Cardenas-Montes, G. Fedak, and F. Araujo. “EDGeS: The Common Boundary Between Service and Desktop Grids.” Parallel Processing Letters (PPL), Special Issue on Grid Architectural Issues: Scalability, Dependability, Adaptability. Volume: 18, Issue: 3, pp. 37–48. September, 2008.

I. Kelley and I. Taylor. “Bridging the Data Management Gap Between Service and Desktop Grids,” in *Proceedings of the 7th International Conference on Distributed and Parallel Systems (DAPSYS 2008)*. Debrecen, Hungary. 3–5 September 2008.

F. Costa, L. Silva, G. Fedak, and I. Kelley. “Optimizing Data Distribution in Desktop Grid Platforms.” Parallel Processing Letters (PPL), Special Issue on Grid Architectural Issues: Scalability, Dependability, Adaptability. Volume: 18, Issue: 3, pp. 391–410. September, 2008.

G. Fedak, H. He, O. Lodygensky, Z. Balaton, Z. Farkas, G. Gombas, P. Kacsuk, R. Lovas, A. C. Marosi, I. Kelley, I. Taylor, G. Terstyanszky, T. Kiss, M. Cardenas-Montes, A. Emmen, and F. Araujo. “EDGeS: A Bridge between Desktop Grids and Service Grids,” in *Proceedings of the Third ChinaGrid Annual Conference (ChinaGrid 2008)*. Dunhuang, China. 20–22 August 2008

I. Kelley and I. Taylor. “Bridging the Data Management Gap Between Service and Desktop Grids.” Distributed and Parallel Systems, In Focus: Desktop Grid Computing, pp. 13–26. Peter Kacsuk, Robert Lovas and Zsolt Nemeth (Editors), Springer, August 2008.

D. Barbalace, D. Talia, I. Kelley, I. Taylor, and C. Mastroianni. "A Data-Sharing Protocol for Desktop Grid Projects." CoreGRID Network of Excellence – Institutes on Knowledge and Data Management & Grid Systems, Tools and Environments. Technical Report TR-0165. August 2008.

F. Costa, L. Silva, I. Kelley, and I. Taylor. "P2P Techniques for Data Distribution in Desktop Grid Computing Platforms." Making Grids Work, Marco Danelutto, Paraskevi Fragopoulou and Vladimir Getov (Editors), Springer, July 2008.

P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor. "Cache-Enabled Super-Peer Overlays for Multiple Job Submission on Grids." Grid Middleware and Services: Challenges and Solutions, pp. 155–169. Domenico Talia, Ramin Yahyapour, and Wolfgang Ziegler (Editors), Springer, July 2008.

F. Costa, L. Silva, G. Fedak, and I. Kelley. "Optimizing the Data Distribution Layer of BOINC with BitTorrent." CoreGRID Network of Excellence – Institute on Architectural Issues: Scalability, Dependability, Adaptability. Technical Report TR-0139. June 2008

M. Cardenas-Montes, A. Emmen, A. C. Marosi, F. Araujo, G. Gombas, G. Fedak, I. Kelley, I. Taylor, O. Lodygensky, P. Kacsuk, R. Lovas, T. Kiss, Z. Balaton, Z. Farkas, and G. Terstyanszky, "EDGeS: BridgingDesktop and Service Grids," in *Proceedings of the Second Iberian Grid Infrastructure Conference (IBERGRID 2008)*. Porto, Portugal. 12–14 May 2008.

F. Costa, L. Silva, G. Fedak, and I. Kelley. "Optimizing the Data Distribution Layer of BOINC with BitTorrent," in *Proceedings of the Second Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2008) held in conjunction with the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Miami, Florida, U.S.A. 18 April 2008

Z. Balaton, Z. Farkas, G. Gombas, P. Kacsuk, R. Lovas, A. C. Marosi, A. Emmen, G. Terstyanszky, T. Kiss, I. Kelley, I. Taylor, O. Lodygensky, M. Cardenas-Montes, G. Fedak, and F. Araujo. "EDGeS: The Common Boundary Between Service and Desktop Grids," in *Proceedings of the CoreGRID Integration Workshop 2008: Integrated Research in Grid Computing*. Crete, Greece. 2–4 April 2008.

F. Costa, L. Silva and I. Kelley. "Using a Content Distribution Network for Data Distribution in Desktop Grid Computing Platforms." Poster at the CoreGRID Integration Workshop 2008: Integrated Research in Grid Computing. Crete, Greece. 2–4 April 2008.

P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor. "Use of P2P Overlays for Distributed Data Caching in Public Scientific Computing." CoreGRID Network of Excellence – Institute on Knowledge and Data Management. Technical Report TR-0112. October 2007.

F. Costa, L. Silva, I. Kelley and I. Taylor. "Peer-To-Peer Techniques for Data Distribution in Desktop Grid Computing Platforms." CoreGRID Network of Excellence – Institute on Architectural Issues: Scalability, Dependability, Adaptability. Technical Report TR-0095. July 2007.

P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor. "Cache-Enabled Super-Peer Overlays for Multiple Job Submission on Grids," in *Proceedings of the CoreGRID workshop on Grid Middleware, in conjunction with ISC07*. Dresden, Germany. 25–26 June 2007.

F. Costa, L. Silva, I. Kelley, and I. Taylor. "P2P Techniques for Data Distribution in Desktop Grid Computing Platforms," in *Proceedings of the CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments (CIW'07)*. Crete, Greece. 12–13 June 2007.

Bibliography

[1] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, Debunking the 100X GPU vs. CPU Myth: an Evaluation of Throughput Computing on CPU and GPU, in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*, 2010, pp. 451–460.

[2] L. G. Roberts, Multiple Computer Networks and Intercomputer Communication, in *Proceedings of the First ACM Symposium on Operating System Principles (SOSP '67)*, 1967, pp. 3.1–3.6.

[3] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computer Infrastructure*. Morgan-Kaufmann, 1999.

[4] I. Foster, C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organization, *The International Journal of High Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.

[5] I. Foster, What is the Grid? - a Three Point Checklist, *GRIDtoday*, vol. 1, no. 6, July 2002. See: *http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf*

[6] I. Foster, The Grid: A New Infrastructure for 21st Century Science, in *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Ltd, 2003, pp. 51–63.

[7] W. Gentzsch, Response to Ian Foster's What is the Grid, Grid Today, August 2002.

[8] New to Grid Computing, IBM Developerworks, January 2004. See: *http://www-106.ibm.com/developerworks/grid/newto/*

[9] M. Litzkow, M. Livny, and M. Mutka, Condor: A Hunter of Idle Workstations, in *Proceedings of 8th International Conference of Distributed Computing Systems,*, 104-111, Ed., June 1988.

[10] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, Condor – A Distributed Job Scheduler, in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.

[11] European Grid Infrastructure (EGI). See: *http://www.egi.eu/*

[12] The Open Science Grid Consortium. See: *http://www.opensciencegrid.org/*

[13] The TeraGrid Project. See: *http://www.teragrid.org*

[14] OxGrid, a Campus Grid for the University of Oxford. See: *http://www.ict.ox.ac.uk/strategy/events/wallom/*

[15] N. Andrade, L. Costa, G. Germóglio, and W. Cirne, Peer-to-Peer Grid Computing with the OurGrid Community, in *Proceedings of the 23rd Brazilian Symposium on Computer Networks (SBRC05)*, 2005.

[16] Amazon Web Services. See: *http://aws.amazon.com/*

[17] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, The Eucalyptus Open-source Cloud-computing System, in *Proceedings of Cloud Computing and Its Applications*, Oct. 2008.

[18] D. Milojicic, I. M. Llorente, and R. S. Montero, OpenNebula: A Cloud Management Tool, *IEEE Internet Computing*, vol. 15, pp. 11–14, 2011.

[19] Amazon Simple Storage Service (S3). See: *http://aws.amazon.com/s3/*

[20] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, Amazon S3 for Science Grids: a Viable Solution? in *Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing (DADC '08)*, 2008, pp. 55–64.

[21] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, Tech. Rep., 1994.

[22] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, Tech. Rep., 1995. See: *http://www.mpi-forum.org*

[23] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud, in *Proceedings of the 2nd International Conference on Cloud Computing (CloudCom)*, 2010, pp. 159–168.

[24] TOP 500 Supercomputing Sites. See: *http://www.top500.org/*

[25] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. Anderson, Cost-Benefit Analysis of Cloud Computing versus Desktop Grids, in *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*, 2009, pp. 1–12.

[26] G. Cocconi and P. Morrison, Searching for Interstellar Communications, *Nature*, vol. 184, no. 4690, 1959.

[27] distributed.net. See: *http://www.distributed.net/*

[28] Great Internet Mersenne Prime Search (GIMPS). See: *http://www.mersenne.org/*

[29] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, SETI@home: An Experiment in Public-Resource Computing, *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, November 2002.

[30] SETI@home. See: *http://setiathome.ssl.berkeley.edu/*

[31] D. Anderson, Public Computing: Reconnecting People to Science, *Presented at the Conference on Shared Knowledge and the Web*, Nov. 2003.

[32] D. Anderson, BOINC: A System for Public-Resource Computing and Storage, in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, 2004, pp. 4–10.

[33] Berkeley Open Infrastructure for Network Computing (BOINC). See: *http://boinc.berkeley.edu/*

[34] BOINC Stats. See: *http://www.boincstats.com/*

[35] Climate@Home. See: *http://www.nasa.gov/offices/ocio/ittalk/08-2010_climate.html*

[36] Einstein@Home. See: *http://einstein.phys.uwm.edu/*

[37] F. Cappello *et al.*, Computing on Large-Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid, *Future Generation Computer Systems* , vol. 21, no. 3, pp. 417–437, 2005.

[38] P. J. Sutton, Searching for Gravitational Waves with LIGO, *Journal of Physics: Conference Series*, vol. 110, no. 6, 2008.

[39] Climateprediction.net. See: *http://www.climateprediction.net/*

[40] D. S. Myers, A. L. Bazinet, and M. P. Cummings, Expanding the Reach of Grid Computing: Combining Globus- and BOINC-based Systems, in *Grids for Bioinformatics and Computational Biology*. Wiley Book Series on Parallel and Distributed Computing, 2008.

[41] Z. Balaton, G. Gombas, P. Kacsuk, A. Kornafeld, J. Kovacs, A. C. Marosi, G. Vida, N. Podhorszki, and T. Kiss, SZTAKI Desktop Grid: a Modular and Scalable Way of Building Large Computing Grids, *Parallel and Distributed Processing Symposium, International*, p. 475, 2007.

[42] M. Silberstein, Building an Online Domain-Specific Computing Service over Non-dedicated Grid and Cloud Resources: The Superlink-Online Experience, in *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011)*, 2011, pp. 174–183.

[43] Enabling Grids for E-sciencE (EGEE). See: *http://www.eu-egee.org/*

[44] G. A. Stewart, D. Cameron, G. A. Cowan, and G. McCance, Storage and Data Management in EGEE, in *Proceedings of the Fifth Australasian Symposium on ACSW Frontiers (ACSW '07)*, 2007, pp. 69–77.

[45] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, The Globus Striped GridFTP Framework and Server, in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*, 2005.

[46] J. Novotny, S. Tuecke, and V. Welch, An Online Credential Repository for the Grid: MyProxy, in *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*, 2001.

[47] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell Agnello, kos Frohner, A. Gianoli, K. Lrentey, and F. Spataro, VOMS, an Authorization System for Virtual Organizations, in *Grid Computing, First European Across Grids Conference*, ser. Lecture Notes in Computer Science, F. F. Rivera, M. Bubak, A. G. Tato, and R. Doallo, Eds., vol. 2970. Springer, 2003, pp. 33–40.

[48] gLite - Lightweight Middleware for Grid Computing. See: *http://glite.web.cern.ch/glite/*

[49] e-ScienceTalk. See: *http://www.e-sciencetalk.org/*

[50] Enabling Desktop Grids for e-Science (EDGeS). See: *http://www.edges-grid.eu/*

[51] Z. Balaton, Z. Farkas, G. Gombas, P. Kacsuk, R. Lovas, A. Marosi, G. Terstyanszky, T. Kiss, O. Lodygensky, G. Fedak, A. Emmen, I. Kelley, I. Taylor, M. Cardenas-Montes, and F. Araujo, EDGeS: The Common Boundary Between Service And Desktop Grids, in *Grid Computing*, S. Gorlatch, P. Fragopoulou, and T. Priol, Eds. Springer, 2008, pp. 37–48.

[52] T. Kiss, G. Szmetanko, D. Farkas, G. Terstyanszky, and P. Kacsuk, Porting Applications to a Combined Desktop Grid/Service Grid Platform using the EDGeS Application Development Methodology, in *Proceedings of the 4th International Workshop on Distributed Cooperative Laboratories: Instrumenting the Grid (INGRID 09)*, 2009.

[53] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, A Resource Management Architecture for Metacomputing Systems, in *Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/SPDP '98)*, 1998, pp. 62–82.

[54] E. Urbah, P. Kacsuk, Z. Farkas, G. Fedak, G. Kecskemeti, O. Lodygensky, C. A. Marosi, Z. Balaton, G. Caillat, G. Gombás, A. Kornafeld, J. Kovács, H. He, and R. Lovas, EDGeS: Bridging EGEE to BOINC and XtremWeb, *Journal of Grid Computing*, vol. 7, no. 3, pp. 335–354, 2009.

[55] G. Caillat, O. Lodygensky, E. Urbah, G. Fedak, and H. He, Towards a Security Model to Bridge Internet Desktop Grids and Service Grids, in *Euro-Par 2008 Workshops - Parallel Processing*, ser. Lecture Notes in Computer Science, E. César, M. Alexander, A. Streit, J. Träff, C. Cérin, A. Knüpfer, D. Kranzlmüller, and S. Jha, Eds. Springer, 2009, vol. 5415, pp. 247–259.

[56] P. Kacsuk, Z. Farkas, and G. Fedak, Towards Making BOINC and EGEE Interoperable, in *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, 2008, pp. 478–484.

[57] T. Kiss, I. Kelley, and P. Kacsuk, Porting Computation and Data Intensive Applications to Distributed Computing Infrastructures Incorporating Desktop Grids, in *Proceedings of the International Symposium on Grids & Clouds (ISGC 2011)*, 2011.

[58] C. Shirky, What Is P2P... And What Isn't, November 2000. See: *http://openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html*

[59] A. Oram, From P2P to Web Services: Addressing and Coordination, April 2004. See: *http://www.xml.com/pub/a/2004/04/07/p2p-ws.html*

[60] I. Taylor, *From P2P to Web Services and Grids: Peers in a Client/Server World*. Springer, October 2004.

[61] B. Carlsson and R. Gustavsson, The Rise and Fall of Napster - An Evolutionary Approach, in *Proceedings of the 6th International Computer Science Conference on Active Media Technology (AMT '01)*, 2001, pp. 347–354.

[62] S. Iyer, A. Rowstron, and P. Druschel, Squirrel: A Decentralized Peer-to-Peer Web Cache, in *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC '02)*, 2002.

[63] GNU General Public License. Version 3, June 2007. See: *http://www.gnu.org/licenses/gpl.html*

[64] Napster. See: *http://www.napster.co.uk/*

[65] B. Cohen, Incentives Build Robustness in BitTorrent, in *Proceedings of the Workshop on Economics of Peer-to-Peer Systems (P2PEcon '03)*, June 2003.

[66] B. Wei, G. Fedak, and F. Cappello, Scheduling Independent Tasks Sharing Large Data Distributed with BitTorrent, in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (GRID '05)*, 2005, pp. 219–226.

[67] B. Nicolae, G. Antoniu, and L. Bouge, Blobseer: Efficient data management for data-intensive applications distributed at large-scale, in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, april 2010, pp. 1 –4.

[68] J. Dean and S. Ghemawat, Mapreduce: simplified data processing on large clusters, *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. See: *http://doi.acm.org/10.1145/1327452.1327492*

[69] A. Joseph, J. Kubiatowicz, L. Huang, S. Rhea, J. Stribling, and B. Zhao, Tapestry: A Resilient Global-Scale Overlay for Service Deployment, *IEEE Journal on Selected Areas in Communications*, vol. 22, 2004.

[70] A. I. T. Rowstron and P. Druschel, Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, 2001, pp. 329–350.

[71] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, A Scalable Content-Addressable Network, in *Proceedings of the ACM SIGCOMM Conference*, 2001.

[72] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications, in *Proceedings of the ACM SIGCOMM Conference*, 2001.

[73] D. Li, X. Lu, and J. Su, Graph-Theoretic Analysis of Kautz Topology and DHT Schemes, in *Network and Parallel Computing*, ser. Lecture Notes in Computer Science, H. Jin, G. Gao, Z. Xu, and H. Chen, Eds. Springer, 2004, vol. 3222, pp. 308–315.

[74] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. G. s, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, OceanStore: An Architecture for Global-Scale Persistent Storage, *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[75] I. Clarke, S. G. Miller, O. Sandberg, B. Wiley, and T. W. Hong, Protecting Free Expression Online with Freenet, *IEEE Internet Computing*, pp. 40–49, February 2002.

[76] I. Foster, A. Iamnitchi, and R. Matei, Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design, *IEEE Internet Computing*, vol. 6, pp. 50–57, 2002.

[77] J. Liang, R. Kumar, and K. Ross, The KaZaA Overlay: A Measurement Study, in *Proceedings of the 19th IEEE Annual Computer Communications Workshop*, 2004.

[78] KaZaA, 2008. See: *http://www.kazaa.com/*

[79] S. Ghemawat, H. Gobioff, and S.-T. Leung, The Google File System, *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[80] S. S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, N. T. Mineedi, and S. L. Scott, FreeLoader: Scavenging Desktop Storage Resources for Scientific Data, in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05)*, 2005.

[81] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment, in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002, pp. 1–14.

[82] W. J. Bolosky, J. R. Douceur, and J. Howell, The Farsite Project: A Retrospective, *SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 17–26, April 2007.

[83] C. A. Thekkath, T. Mann, and E. K. Lee, Frangipani: A Scalable Distributed File System, in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, 1997, pp. 224–237.

[84] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh, stdchk: A checkpoint storage system for desktop grid computing, in *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, june 2008, pp. 613 –624.

[85] F. Costa, I. Kelley, L. Silva, and I. Taylor, Peer-To-Peer Techniques for Data Distribution in Desktop Grid Computing Platforms, in *Making Grids Work*, M. Danelutto, P. Fragopoulou, and V. Getov, Eds. Springer, 2008.

[86] C. Mastroianni, D. Talia, and O. Verta, A Super-Peer Model for Resource Discovery Services in Large-Scale Grids, *Future Generation Computer Systems*, vol. 21, no. 8, pp. 1235–1248, 2005.

[87] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. A. Felber, A. Al Hamra, and L. Garcés-Erice, Dissecting BitTorrent: Five Months in a Torrent's Lifetime, in *Passive and Active Network Measurement*, ser. Lecture Notes in Computer Science, C. Barakat and I. Pratt, Eds. Springer, 2004, vol. 3015, pp. 1–11.

[88] D. Anderson and G. Fedak, The Computational and Storage Potential of Volunteer Computing, in *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, 2006, pp. 73–80.

[89] F. Costa, L. M. Silva, G. Fedak, and I. Kelley, Optimizing Data Distribution in Desktop Grid Platforms, *Parallel Processing Letters*, vol. 18, no. 3, pp. 391–410, 2008.

[90] L. Cherkasova and J. Lee, FastReplica: Efficient Large File Distribution within Content Delivery Networks, in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[91] IETF Public Key Infrastructure Working Group. See: *http://www.ietf.org/html.charters/pkix-charter.html*

[92] K. Berket, A. Essiari, and A. Muratas, PKI-Based Security for Peer-to-Peer Information Sharing, in *Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, 2004, pp. 45–52.

[93] J. E. Altman, PKI Security for JXTA Overlay Networks, IAM Consulting, Inc., Tech. Rep., 2003.

[94] T. Barton, J. Basney, T. Freeman, T. Scavo, F. Siebenlist, V. Welch, R. Ananthakrishnan, B. Baker, M. Goode, and K. Keahey, Identity Federation and Attribute-based Authorization through the Globus Toolkit, Shibboleth, Gridshib, and MyProxy, in *Proceedings of the 5th Annual PKI R&D Workshop*, 2006.

[95] F. Costa, L. M. Silva, I. Kelley, and G. Fedak, Optimizing the Data distribution Layer of BOINC with BitTorrent, in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–8.

[96] D. Barbalace, P. Cozza, D. Talia, and C. Mastroianni, A P2P Job Assignment Protocol for Volunteer Computing Systems, Institute on Knowledge and Data Management and Institute on Architectural Issues: Scalability, Dependability, Adaptability, CoreGRID - Network of Excellence, Tech. Rep. TR-0117, 2007.

[97] P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor, Cache-Enabled Super-Peer Overlays for Multiple Job Submission on Grids, in *Grid Middleware and Services: Challenges and Solutions*, D. Talia, R. Yahyapour, and W. Ziegler, Eds. Springer, 2008.

[98] P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor, Use of P2P Overlays for Distributed Data Caching in Public Scientific Computing, Institute on Knowledge and Data Management, CoreGRID Network of Excellence, Tech. Rep. TR-0112, October 2007.

[99] C. Mastroianni, P. Cozza, D. Talia, I. Kelley, and I. Taylor, A Scalable Super-Peer Approach for Public Scientific Computation, *Future Generation Computer Systems*, vol. 25, no. 3, pp. 213–223, Mar. 2009.

[100] I. Wang, P2PS (Peer-to-Peer Simplified), in *Proceedings of 13th Annual Mardi Gras Conference – Frontiers of Grid Applications and Technologies*, 2005, pp. 54–59.

[101] JXTA: The Language and Platform Independent Protocol for P2P Networking. See: *http://jxta.kenai.com/*

[102] I. Kelley and I. Taylor, Bridging the Data Management Gap Between Service and Desktop Grids, in *Distributed and Parallel Systems. In Focus: Desktop Grid Computing*. Springer, 2008, pp. 13–26.

[103] I. Kelley and I. Taylor, A Peer-to-Peer Architecture for Data-Intensive Cycle Sharing, in *Proceedings of the First International Workshop on Network-Aware Data Management (NDM '11)*, 2011, pp. 65–72.

[104] D. Bickson and D. Malkhi, The Julia Content Distribution Network, in *Proceedings of the 2nd Conference on Real, Large Distributed Systems (WORLDS '05)*, vol. 2, 2005, pp. 37–41.

[105] P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor, Cache-Enabled Super-Peer Overlays for Multiple Job Submission on Grids, in *Proceedings of the CoreGRID Workshop on Grid Middleware*, 2007.

[106] D. Barbalace, D. Talia, I. Kelley, I. Taylor, and C. Mastroianni, A Data Sharing Protocol for Desktop Grid Projects, Institutes on Knowledge and Data Management & Grid Systems, Tools and Environments, CoreGRID - Network of Excellence, Tech. Rep. TR-0165, 2008.

[107] The Network Simulator – NS-2. See: *http://www.isi.edu/nsnam/ns/*

[108] I. Taylor, I. Downard, B. Adamson, and J. Macker, AgentJ: Enabling Java NS-2 Simulations for Large Scale Distributed Multimedia Applications, in *Proceedings of the Second International Conference on Distributed Frameworks for Multimedia (DFMA 2006)*, 2006.

[109] NorduGrid: Advanced Resource Connector (ARC). See: *http://www.nordugrid.org/arc/*

[110] UNICORE: UNiform Interface to COmputing REsources. See: *http://www.unicore.org*

[111] The Attic File System. See: *http://www.atticfs.org/*

[112] GitHub. See: *http://www.github.com/*

[113] European Desktop Grid Initiative (EDGI). See: *http://www.edgi-project.eu/*

[114] The application/json Media Type for JavaScript Object Notation (JSON), 2006. See: *http://tools.ietf.org/html/rfc4627*

[115] The Transport Layer Security (TLS) Protocol Version 1.2, 2008. See: *http://tools.ietf.org/html/rfc5246*

[116] A Universally Unique IDentifier (UUID) URN Namespace, 2005. See: *http://www.ietf.org/rfc/rfc4122.txt*

[117] A. Harrison, Peer-to-Grid Computing, Ph.D. dissertation, School of Computer Science & Informatics, Cardiff University, 2007.

[118] A. Harrison, I. Kelley, K. Mueller, M. Shields, and I. Taylor, Workflows Hosted In Portals, in *Proceedings of the UK eScience All Hands Meeting (AHM '07)*, 2007.

[119] Workflows Hosted in Portals (WHIP). See: *http://www.omii.ac.uk/wiki/WHIP*

[120] Restlet - RESTful Web Framework for Java. See: *http://www.restlet.org/*

[121] D6.1 Supplementary Report on: Data Distribution Paths from SGs to DGs, European Desktop Grid Initiative, Tech. Rep., January 2011.

[122] G. Fedak, H. He, O. Lodygensky, Z. Balaton, Z. Farkas, G. Gombas, P. Kacsuk, R. Lovas, A. C. Marosi, I. Kelley, I. Taylor, G. Terstyanszky, T. Kiss, M. Cardenas-Montes, A. Emmen, and F. Araujo, EDGeS: A Bridge between Desktop Grids and Service Grids, in *Proceedings of the The Third ChinaGrid Annual Conference (ChinaGrid 2008)*, 2008, pp. 3–9.

[123] Integrated ARC, Desktop Grid, and Eucalyptus bridge, European Desktop Grid Initiative, Tech. Rep., January 2011.

[124] Computing Resource Execution And Management (CREAM). See: *http://grid.pd.infn.it/cream/*

[125] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, The Hadoop Distributed File System, in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)*, 2010, pp. 1–10.

[126] The Attic Proxy (libafs). See: *http://www.atticfs.org/libafs/*

[127] Uniform Resource Identifier (URI): Generic Syntax [RFC3986], Network Working Group, 2005. See: *http://www.ietf.org/rfc/rfc3986.txt*

[128] B. Maso, A New Era for Java Protocol Handlers, White Paper, August 2000.

[129] I. Kelley, Attic Data Distribution Framework, *Presentation at the 7th BOINC Workshop*, August 2011. See: *http://boinc.berkeley.edu/trac/wiki/WorkShop11*

[130] A. Elwaer, A. Harrison, I. Kelley, and I. Taylor, Attic: A Case Study for Distributing Data in BOINC Projects, in *Proceedings of the Fifth Workshop on Desktop Grids and Volunteer Computing Systems (PCGrid 2011)*, 2011.

[131] G. Fedak, H. He, and F. Cappello, BitDew: a programmable environment for large-scale data management and distribution, in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[132] J. Kovacs, F. Araujo, S. Boychenko, M. Keller, and A. Brinkmann, Monitoring unicore jobs executed on desktop grid resources, in *Proceedings of the 36th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2012.

[133] A. Harrison and I. Taylor, Dynamic Web Service Deployment Using WSPeer, in *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, 2005, pp. 11–16.

[134] Jersey (JSR 311 Reference Implementation). See: *http://jersey.java.net/*

[135] Dropbox. See: *http://www.dropbox.com/*

[136] Filesystem in Userspace (FUSE). See: *http://fuse.sourceforge.net/*

[137] CoreGRID: European Research Network on Foundations, Software Infrastructures and Applications for Large-Scale Distributed, GRID and Peer-to-Peer Technologies. See: *http://www.ercim.eu/activity/projects/coregrid.html*

[138] I. Taylor, E. Al-Shakarchi, and S. D. Beck, Distributed Audio Retrieval using Triana (DART), in *Proceedings of the International Computer Music Conference (ICMC)*, 2006, pp. 716–722.

[139] E. Al-Shakarchi, C. Pasquale, A. Harrison, C. Mastroianni, M. Shields, D. Talia, and I. Taylor, Distributing Workflows over a Ubiquitous P2P Network, *Scientific Programming – Dynamic Computational Workflows: Discovery, Optimization and Scheduling*, vol. 15, no. 4, pp. 269–281, 2007.

[140] A. Elwaer, I. Taylor, and O. Rana, Preference Driven Server Selection in Peer-2-Peer Data Sharing Systems, in *Proceedings of the Fourth International Workshop on Data-Intensive Distributed Computing (DIDC '11)*, 2011, pp. 9–16.

[141] A. Elwaer, I. Taylor, and O. Rana, Optimizing Data Distribution in Volunteer Computing Systems using Resources of Participants, *Scalable Computing: Practice and Experience*, vol. 12, no. 2, 2011.