SCALABLE, FAULT-TOLERANT MANAGEMENT OF GRID SERVICES: APPLICATION TO MESSAGING MIDDLEWARE

Harshawardhan Gadgil

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

May 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Geoffrey Fox, Ph.D.
(Principal Advisor)

Randall Bramley, Ph.D.

Beth Plale, Ph.D.

Edward Robertson, Ph.D.

April 5, 2007

Copyright © 2007
Harshawardhan Gadgil
Department of Computer Science
Indiana University
ALL RIGHTS RESERVED

To my Parents, with sincere love and respect.

ACKNOWLEDGEMENTS

During the course of present research work, I have benefited from support, encouragement and guidance from my family, mentors, colleagues and friends. I owe them a great deal of gratitude and wish to express my sincere thanks.

I'm indebted to my advisor, Prof. Geoffrey Fox. Without his inspiring guidance, invaluable advice, constant encouragement and confidence in my abilities, this dissertation would not have been possible. I would like to express my deepest gratitude for all his help.

I would especially like to thank Dr. Shrideep Pallickara. I've had the opportunity to work closely with him while contributing to the NaradaBrokering project and during this dissertation. I've learnt a lot from his impeccable attitude towards research while endless discussions with him have helped shape a lot of my ideas. I would also like to thank Dr. Marlon Pierce who provided guidance when the going seemed tough during various bottlenecks encountered in the last four years. I am also thankful to my committee members: Prof. Edward Robertson, Prof. Beth Plale and Prof. Randall Bramley, for their useful comments.

Through my graduate school, I've been fortunate to have the support and encouragement from many colleagues and friends at Indiana University. I would like to thank my lab mates: Sangyoon Oh, Mehmet Aktas, Hasan Bulut, Galip Aydin, Ahmet Sayar and Beytullah Yildiz in the Community Grids Lab at Indiana University. My friends: Pavan, Amit, Sidharth, Ketan, Deep, Sharat and Sumit have been around, especially during the final phase of Ph.D. Their company has been especially helpful when unwinding every day's work.

I owe a great deal of thanks to my family for supporting and encouraging me at every step of my Ph.D. work. My parents have been an immense source of inspiration and have encouraged me every step of the way. They deserve much credit for all my accomplishments. For my wife Prajakta, no words of praise can suffice. She has been my pillar of strength during the grueling

years of my Ph.D. I cannot thank her enough for all the support and encouragement that she has provided. I would not have made it this far without her.

Finally, I would like to thank the entire Computer-Science department and the CGL staff who have in one way or other helped make the last six years a fun and rewarding experience. Thank you.

ABSTRACT

Scalable, Fault-tolerant Management of Grid Services: Application to Messaging Middleware

By

Harshawardhan Gadgil

Doctor of Philosophy in Computer Science

Indiana University, Bloomington

Prof. Geoffrey C. Fox, Chair

The service-oriented architecture has come a long way in solving the problem of reusability of existing software resources. As Service-based architectures emerge, *management* of the application which comprises of a large number of distributed services becomes even more difficult as resources *appear, move and disappear* across the network. Further, the application components may span disparate network boundaries, which add a variety of constraints such as network policies, blocked transports and authentication requirements. Services exist on different platforms and are written in different languages. This makes use of any single management technology inefficient and promotes non-interoperability.

In this thesis, we present a management architecture that combines *publish-subscribe* and *service-oriented computing* principles for managing a set of distributed entities. The use of service-oriented architecture adds interoperability to the management process. The proposed system adopts a distributed hierarchical architecture to achieve scalability. We show that the architecture is tolerant to failures in the management framework itself and can be extended to provide user-level fault-tolerance of managed resources by implementing appropriate policies. Finally, we

present an empirical evaluation of the system and demonstrate that the proposed architecture adds an acceptable number of additional resources required for providing fault-tolerance of various components in the system.

TABLE OF CONTENTS

LIST OF FIGURES

APPENDIX A

LIST OF TABLES

APPENDIX B

Chapter 1. Introduction

1.1. Introduction

Computing Systems are constantly undergoing revolution. From the early inception of modern computers in 1945, until about 1985, computers were large and expensive. As a result, most organizations had only a handful of computers and no means to connect them, effectively these machines operated independently.

Starting mid-80s, this scenario changed drastically. Advancement in technology made powerful CPUs common. These provided the power of mainframes but at a fraction of the price. The size of computing devices went down from a roomful to one that could fit in the palm of human hand. The second development was the invention of high speed computer networks which allowed multitude of machines to be connected together. Large amounts of data could be moved between them at a rate in excess of 100 million bits/sec. The result of these technologies is that it has now

become not only feasible but also easy to connect computing systems composed of a large number of computers connected by high-speed network links.

A side-effect of the advancement was that different computing systems were introduced. Some were made too simple to carry out the duties they were supposed to perform while others were made too complex. The cost of building and maintaining them rocketed, not to mention the nearly impossible task of integrating different systems together. As more and more software systems are built, similar situations and patterns appear. Naturally, we want to reuse the functionality of existing systems rather than building them from scratch. This led to the idea of a Service Oriented Architecture (SOA) [1, 2], which is an architectural style for loosely coupling interacting software agents. A SOA provides an interoperable platform that helps maximize reusability of existing software resources. While it is clear that SOA and its current implementation, Web services [3], will have profound impact on the next generation of distributed systems, many aspects of this platform still require significant research and development.

The emerging trend of building distributed applications is to reuse distributed services. A successful distributed application benefits from properly managed (configured, deployed and monitored) services while effective management of these distributed services benefits from well-instrumented computer software and hardware. One challenge is that the technologies used to deploy, configure, secure, monitor and control resources have evolved independently of each other. For instance, many network devices use SNMP (Simple Network Management Protocol) [4], Java applications use JMX (Java Management eXtensions) [5], while servers implement management using CIM (Common Information Model) [6] or WBEM (Web-based Enterprise Management) [7].

The Web services community has addressed this challenge by adopting a SOA using Web service technology to provide *flexible* and *interoperable* management protocols. The goal of the SOA is not

to replace existing protocols but to successfully extend management frameworks to permit more effective integration with existing management systems. The *flexibility* comes from the ability to quickly adapt to rapidly changing requirements. The *interoperability* comes from the use of XML based interactions that facilitate implementations in different languages, running on different platforms and over multiple transports.

This dissertation is motivated by understanding the need for a *Distributed Management Framework* that effectively combines *"Management operations in a distributed system"* and *"SOA principles"*; evaluating the system design parameters in terms of scalability, fault-tolerance, performance, interoperability, generality and usability; and evaluating how these factors influence the overall infrastructure cost in terms of additional resources.

1.2. The concept of Resource Management

1.2.1. Definition of term "Resource"

Before we discuss the aspects of resource / service management, we would like to clarify the use of term *Resource* as used in this thesis. Distributed applications are composed of components which are entities on the network. We consider a specific case of distributed applications where these entities can be controlled by zero or modest state. We define modest state as being one which can be exchanged using very few messages (typically 1 message exchange). These entities in turn can initialize and control components with much higher state. Such components could be hardware (e.g. network, CPU, memory) or software (e.g., file-systems, databases or services). We consider the combination of such an entity and the component associated with it as a manageable resource. Note that we do not imply any relation to other definitions of the term "resource" elsewhere in literature (e.g. WS – Resource as defined by WS – Resource Framework [8]).

Since the primary target of the implementation of this thesis is management of software services, we consider a software service as a manageable resource.

1.2.2. Aspects of Resource / Service Management

The primary goal of *Resource Management* is the efficient and effective deployment of available resources. A distributed system is composed of a large number of widely distributed resources. Management can be defined as the function that aims at "*Maintaining the system's ability to provide its specified services with a prescribed quality of service*". Resource management can be divided into two broad domains: one that primarily deals with efficient resource utilization, and another that deals with resource administration activities such as configuration, deployment and monitoring.

In the first category, resource management deals with resource allocation and scheduling, the goal being sharing resources fairly while maintaining optimal resource utilization. For example, an operating system [9] provides resource management by providing fair overall resource sharing among users through various means such as process scheduling and memory management. Condor [10] provides a specialized job management for compute-intensive jobs. GRAM (Grid Resource Allocation Manager) [11] provides an interface for requesting and using remote system resources for the execution of jobs.

The second category deals with appropriately configuring and deploying resources (services) while maintaining a valid run-time configuration according to some user-defined criteria. In this case management has *static* (configuring and bootstrapping the system components) and *dynamic* (runtime monitoring and event handling) aspects. In general terms, management can be viewed as a control activity that involves detecting events that alter the ability of an administered system to perform its function and reacting to these events by trying to restore this ability. These administration operations may be summarized as follows[1]:

1. Configuration and lifecycle operations such as CREATE, DELETE

[1] http://devresource.hp.com/drc/slide_presentations/wsdm/index.jsp

2. Handling runtime events
3. Monitoring status and performance
4. Maintaining system resources according to user defined criteria

This dissertation addresses the challenges faced in resource administration.

1.3. Motivation

Tshe phenomenal progress of technology has driven the deployment of increasing number of devices ranging from RFID devices to supercomputers. These devices are widely deployed, spanning corporate administrative boundaries. Deployment areas are typically protected by firewalls, thus limiting access to resources. Further, the low cost of hardware has made replication a cost-effective approach to fault-tolerance especially if software replication is used. These factors have contributed to the increasing complexity of today's applications which are composed of ever increasing number of resources: hardware (hard-drives, CPUs, networks) and software (services). Management is required for maintaining a properly running application; however, existing approaches have shown limitations in successfully managing such large scale systems.

Firstly, as the size of systems and applications increases (in terms of number of hardware components, software components and geographical scale) it is certain that some parts of application components will fail. An analysis of the causes of failures of Internet services [12] shows that most of the service's downtime may be attributed to management errors (e.g. wrong configuration) and that software failures come second. However, few efforts have yet been devoted to remedy this situation.

Secondly, administration tasks have mainly been performed by persons. A great deal of the knowledge needed for administration tasks is not formalized and is part of the administrators' know-how and experience. As the size and complexity of the systems and applications are

increasing, the costs related to administration are taking up a major part of the total information processing budgets, and the difficulty of the administration tasks tends to approach the limits of the administrators' skills. The traditional manager-agent model which is the base of widely used management protocols and frameworks (such as SNMP and CMIS/CMIP) is also showing its inability to cope with the current highly dynamic managed systems.

These constraints have motivated a new approach, the so-called *Autonomic Computing* [13] movement, where a significant part of management-related functions is performed automatically with minimal human intervention. Autonomic computing aims at providing systems and applications with the following self-management capabilities, namely: Self-configuration (automatic configuration according to a specified policy), Self-optimization (continuous performance monitoring), Self-healing (detecting defects and failures, and taking corrective actions) and Self-protection (taking preventive measures and defending against malicious attacks). Several research projects [14] are active in this area.

Thirdly, different types of resources in a system require different resource-specific management frameworks. As we have discussed before, the resource management frameworks for different types of resources have evolved independently. This complicates the application implementation by requiring the use of different proprietary technologies for managing different types of resources or using ad-hoc solutions to interoperate between different management protocols.

Finally, centrally managing a distributed application poses many problems. Core problem is scalability. Centralized systems are also vulnerable to a single point of failure. This motivates the need for a distributed management infrastructure. We posit that *"The management framework itself must be tolerant of failures to provide a successful management framework"*.

These factors have motivated the need for a distributed management framework. We envisage a generic management framework that is capable of managing any type of resource. By implementing interoperable management protocols we can effectively integrate existing

management systems. Such a system must be autonomous in providing this functionality with minimum user interaction. One of the chief characteristics of an autonomous system is to provide *self-healing* that automatically handles failures within the system.

We now provide a summary of the desired characteristics of the management framework:

1.3.1. Fault-tolerance

As systems span wide networks they become difficult to maintain and resource failure is norm. Resource failure could be a result of the actual resource failing or a result of failure of some related component such as the network making the resource inaccessible. In such a case, it may be required to instantiate a new resource[2] to take over the functionality of failed resource. Finally, the resource managers themselves might fail making resource management ineffective.

To account for failure scenarios within the management framework, recovery from failures must be facilitated. Typically this would be done by re-instantiating a copy of the failed management framework component which takes over the functionality of the failed component. This, however, leads to possible inconsistencies as failures cannot be reliably distinguished from slowed processes. The schemes outlined for tackling failures must take into account such issues to avoid possible race conditions.

1.3.2. Scalability

As the number of manageable resources increases, the management framework must scale to accommodate management of the additional resources. For example, LHC Grid [15] is a system with a large number of manageable resources. These resources include a large number of CPU nodes (in excess of 10000) for computation, disk nodes for storage and tape servers for access to mass storage among others. Recent web-service offerings from Amazon [16] such as Amazon's

[2] This would be TRUE for software resources such as services

EC2 (*Elastic Compute Cloud*) involves running applications on as many or as few systems as required and dynamically resizing the compute capacity. This represents a set of widely distributed resources constantly in a state of flux, as the compute capacity goes up and down.

Further, the management framework represents additional components that are introduced in the system. Addition of these components (Ref: Section 3.1) is necessary to provide the various features such as fault-tolerance and interoperability. The management framework must scale in terms of these additional components required.

1.3.3. Performance

Runtime events are generated by resources and the management system takes a finite amount of time to respond to faults. The challenge is to achieve acceptable performance in terms of recovery from failure and responsiveness to faults as the number of manageable resources and the additional components required increases. In this thesis, we primarily focus on recovery from failure in terms of responding to run-time events.

1.3.4. Interoperability

Resources may exist on different platforms and may be written in different languages. This promotes use of proprietary solutions for management. Consider for example, *Windows Management Instrumentation* (WMI) [17] and *Java Management eXtensions* (JMX) [5]. These frameworks have been quite successfully implemented. However, these frameworks are not interoperable and thus it limits the applicability of any of these systems for management of resources in heterogeneous systems or platforms. To address interoperability, we leverage service-oriented principles and enable management by implementing a Web-service based management protocol.

1.3.5. Generality

Resource management framework must be generic. This means that it should equally apply to all types of resources i.e., hardware or software resources. Resource-specific management would still be required to be defined by providing a resource-specific management modules and resource-specific policies while the framework provides the basic scalability and fault-tolerance.

1.3.6. Usability

The architecture must be usable in terms of autonomous operation provided by the framework. The framework provides autonomous operation by appropriately instantiating failed management components with minimum user interaction.

1.4. Use Cases

Resources differ in their use and have widely varying characteristics. These characteristics determine the management requirements of the resource. We now describe a few sample use cases that are illustrative of the applicability of the management framework to systems with different resource requirements.

Resources not only vary in their use but also in their capabilities: A typical example would be a slow vs. fast hard disk. One would configure an application such that a slower hard disk would store unvarying data that requires minimum disk access such as application code while a faster hard disk would be configured to be used as high performance storage for continuously varying data such as swap space. Such a configuration is important for performance reasons.

Resources are of completely different types and requirements: A critical infrastructure application such as a Servo Grid Framework [18] consists of services such as a workflow service [19] for executing workflows, a context database service [20] for storing system state and services that integrate local and remote map services (Web Feature Service [21] and Web Map Service

[22]). Each of these services requires application-specific configuration and lifecycle management. Failures are normal and a critical requirement of real-time applications is to keep application components always up and running. Thus, the management framework would also monitor the system for faults and take corrective actions (e.g. create alternate instances of services if a previous instance fails), thus providing continuous and autonomous operation.

Resource need and availability is in a state of flux: Consider for example, an audio/video collaboration system such as GlobalMMCS [23]. This system relies on a distributed messaging substrate for routing audio/video packets to participants. As participants join and leave, the messaging substrate must be appropriately managed to adjust to varying load. Further, runtime metrics provided via monitoring and possible failures within the substrate would require an administrator to dynamically adjust substrate characteristics such as transports (e.g. TCP vs. UDP).

Resources are widely distributed as scale of application increases: As we build bigger systems, these individual resources get widely dispersed. Network constraints such as security policies and presence of firewalls and NAT devices limit access to resources. As we have previously noted, in case of software resources (services) different platforms and language implementations make it harder to use a single technology to manage the variety of resources.

1.5. Research Issues

In this thesis, we describe the architecture design and implementation of a distributed management framework that addresses the desired features mentioned in Section 1.3. We have thoroughly analyzed the system to determine how the system would respond and have presented benchmarks on a typical resource. A major goal of this dissertation is to formulate how much additional software infrastructure is required to manage a set of resources with respect to the number of resources being managed.

We now summarize the research issues we plan to address in this thesis:-

1. How can we build a fault-tolerant management architecture?
2. How can the management framework be made scalable?
3. If such a system can be built, what is the overhead of such a system and is this overhead acceptable?
4. How can we enable global management of resources (i.e. when access to resources may be restricted by presence of firewall and NAT devices)?
5. Can the management system be made interoperable and extensible?

1.6. Summary of Thesis Contributions

This dissertation investigates significant research problems which emerge with a need to uniformly manage a set of distributed resources. We present a novel approach that addresses the issues outlined in Section 1.3 to provide universal management architecture. The work has resulted in the design and implementation of a distributed management framework that is:

1. Tolerant to failures in management framework as well as resource failures by implementing resource-specific policies;
2. Scalable in terms of additional resources required to provide fault-tolerance and performance;
3. Interoperable by use of Web service based management protocol for communication between resources and their respective managers and
4. Generic such that it can be applied equally well to any set of manageable resources

Finally, to demonstrate the use of the management framework, we provide a proof-of-concept implementation for managing a *Grid Messaging Middleware: NaradaBrokering*.

1.7. Thesis Outline

This thesis is organized as follows:-

We present our literature survey in Chapter 2. Here we present an overview of various strategies used in our system such as fault-tolerance and scalability. We also present a brief overview of messaging systems and Web service based management protocols. We conclude the chapter noting the principles applied in our architecture.

We then describe the management architecture in detail in Chapter 3. Here we define the various components of the system and the role they play in the overall architecture.

This is followed by a discussion on the use of a service oriented approach towards management in Chapter 4. Here we introduce the service-oriented management framework based on WS-Management.

In Chapter 5 we analyze the system and discuss the feasibility of our approach. In order to prove that the costs of adding additional resource are acceptable, we answer the question, "*How much Management Infrastructure is required to handle N Resources?*"

Chapter 6 applies the principles towards management of a distributed messaging middleware system (NaradaBrokering). Here we outline the various scenarios in which the prototype was deployed and a discussion on the analysis of system performance in each case.

Finally we present our conclusion and outline the direction of future work in Chapter 7. Here we present answers to the research questions presented in Section 1.5.

Chapter 2. Literature Survey

This thesis leverages well-known principles in distributed systems for achieving scalability and fault-tolerance. In this chapter, we present an overview of the various strategies relevant to our work. We also present examples of systems where these concepts have been implemented. We then discuss existing approaches to management and a discussion of the Web-service based management standards. Finally we present an overview of a messaging middleware: Naradabrokering.

2.1. Fault Tolerance Strategies

Distributed systems have addressed fault-tolerance of application components via strategies such as request-retry, replication and check-pointing. Faults in distributed systems are normal and it is desired that the system continues operation in presence of failures. Fault-tolerance is defined [24]

as the characteristic by which *"A Distributed System can mask the failure occurrence and recover from failure"*.

We now present an overview of some of these schemes.

2.1.1. Replication

Replication schemes provide seamless transfer of control to a new or existing duplicate service instance when failure is detected. Replication can be *Passive* (primary / backup) where only the primary replica processes requests and then state is transferred to other replicas. This helps provide availability in a simple manner. Passive replication does not offer any performance improvement since on failure a backup is promoted to primary which spends some time to restore state from logs.

When performance is an issue and cost of computation is less, *Active* replication is used. In active replication, every replica invokes the operation independently and hence all replicas have the most current state. Thus, on failure, recovery is almost instantaneous. Active replication, however, requires all operations to be carried out at all replicas in the same order. Although techniques such as *Lamport's Timestamps* [25] or using a central coordinator that functions as a *Sequencer* can be used, they suffer from scalability problems. Ref [26] presents a hybrid approach for achieving *Totally Ordered Multicast* in large scale systems.

Distributed databases such as *Oracle* [27] use replication to provide load balancing and high availability in presence of faults. Agents-based technology provides the ability to perform intelligent operations, interactions and cooperation between autonomous components and has also been recognized [28] as a promising technology for managing Web services. Agents provide a fault tolerant substrate for performing system tasks and are very suitable for tasks such as management of distributed resources [29].

Object based distributed systems are an extension of the object-oriented programming systems. As the name suggests, *Distributed Object Computing* allows objects distributed on different computers across a heterogeneous network to interoperate as a unified whole and appear as being local to the application. Communication with remote objects is transparently handled via system specific protocols. Notable efforts are Distributed Component Object Model (DCOM) [30] from Microsoft, Common Object Request Broker Architecture (CORBA) [31] from OMG (Object Management Group) and Java / Remote Method Invocation (Java/RMI) [32] from Javasoft.

DCOM

DCOM addresses fault tolerance via *Automatic Transactions* which allow a developer to specify a series of method invocations (possibly on different objects) that can be grouped into a transaction. A separate transactions manager module called the *Distributed Transactions Coordinator (DTC)* handles the actual implementation of the transactions using standard transaction semantics based on a two-phase commit protocol.

CORBA

CORBA addressed fault-tolerance in version 3.0. The basic approach for dealing with faults is to replicate objects into object groups. Such groups consist of one or more identical copies of same object. Such a group can be referenced as if it were a single object and offers the same interface as the replica it contains. This provides *replication transparency* from the user point of view. Different replication strategies may then be implemented such as active replication, passive replication or quorum based replication. A *Replication Manager* is responsible for creating and managing a group of replicated objects which in turn can be replicated for fault-tolerance.

Java / RMI

The object-oriented nature of Java facilitates code reuse and significantly reduces development time. Further, a wide variety of interfaces and language extensions are available for the Java Virtual Machine (JVM) that makes Java an attractive development platform for most GRID

applications. JVM, however, does not support fault-tolerance. Fault-tolerance is enabled by using systems such as Nomads [33], which modify the JVM to capture the execution state of the application. This is, however, inappropriate for heterogeneous systems where different machines may have different JVMs. Ref [34] describes an approach to make the process of check-pointing independent of the JVM used by modifying the program's bytecode rather than the JVM.

These systems have been successful in their respective areas. However, a crucial limitation is that they are platform specific and not easily interoperable. There are *ad-hoc* solutions to achieve interoperability, such as constructing bridges to translate messages between systems. However, there are other limitations such as fundamental differences between data-types, differing distributed object models and non-friendliness with firewalls and proxy servers. Thus, these systems are unsuitable as a building block to our management framework.

2.1.2. Check-pointing

Check-pointing schemes allow a computation to continue from where it failed rather than re-running the computation. Check-pointing is mainly used in computing systems to store the current state of operation. By switching to an earlier checkpoint, a system can reload the previous state and resume computation from the point of failure. Check-pointing is used in many systems such as *Condor* [10], *XCAT* [35] and MPI based message passing system such as *OpenMPI* [36] and *FT-MPI* [37] to store system state and recover from a previous state after failure has occurred. Besides recovery, check-pointing also enables other features such as process migration [38] which allows a failed process to continue on another machine from the point where it failed.

The main challenge in check-pointing is achieving a *globally consistent* [39] snapshot of the system's state. A survey of various roll-back and recovery protocols can be found in [40]. The main techniques are briefly summarized below:

Independent Check-pointing occurs when all processes maintain local check-points. The main advantage is simplicity and performance. However, such checkpoints may not necessarily be globally consistent. Thus, when processes roll back to the latest checkpoint and if this checkpoint is not globally consistent, another roll back is necessary. Further rolling back is necessary if the last roll back is again inconsistent. This cascaded rollback may lead to what is called the *domino effect*.

Coordinated check-pointing ensures that all processes synchronize to jointly write their state. Although achieving global synchronization is costly in terms of the complexity and time required, the snapshots are automatically globally consistent. Coordinated check-pointing comes in two flavors, *blocking* and *non-blocking*. Blocking algorithms block all check-pointing processes which commit to automatically achieve a globally consistent snapshot. Ref [41] and [42] provide details on blocking coordinated check-pointing implementation. A non-blocking coordinated check-pointing algorithm that uses application-level check-pointing is presented in [43].

2.2. Scalability Strategies

Scalability is a desirable property of a system, a network or a process, which indicates its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged. Scalability can be measured in various dimensions such as *load scalability* (easily expand or contract resource pool to accommodate heavier or lighter loads), *geographical scalability* (maintain usability regardless of resource locations) and *administrative scalability* (easily use and manage a single system irrespective of number of organizations using it).

One of the strategies for improving scalability in a large scale system is to use *asynchronous communication*. The principle idea is to hide communication latencies by allowing multiple requests to be made. When a response arrives, a special handler continues computation of a previously issued request.

Another important scaling technique is via *distribution*. Distribution involves splitting a component into smaller parts and subsequently spreading those parts across the system. A popular system that uses a hierarchical distribution is the DNS (Domain Name System) [44]. The DNS hierarchically organizes the namespace into a tree of domains which are further divided into non-overlapping zones. The names in each zone are handled by a single name server. Hierarchical distribution has also been extensively used in monitoring systems such as SNMP [4] MonALISA [45] and Astrolabe [46].

Finally load distribution techniques such as *replication* aid scaling by helping prevent performance degradation. In geographically distributed systems communication latency can be avoided by satisfying requests from a nearby resource. *Caching*, a special form of replication, is used (typically by clients of resources) that mirrors resource's state locally. This, however, leads to consistency problems and it is up to the resource and the client to determine the degree of inconsistency that the system can tolerate.

2.3. Management Systems

Various system specific management architectures have been developed and have been quite successful in their areas. Examples include SNMP (Simple Network Management Protocol) [4] CMIP [47] and CIM [6]. SNMP defines an application layer protocol that facilitates exchange of management information among network devices. SNMP agents gather data which is aggregated using a tree based hierarchy. This information can further be queried and integrated via a variety of distributed monitoring frameworks. SNMP uses a registry of monitored objects such as CPU, router, bridge, printers in Management Information Base. Thus, component-wise SNMP is similar to the architecture presented in this thesis. However, SNMP deals only with network resources. Lack of security reduces SNMP to a monitoring facility rather than a management facility.

As discussed in Section 1.2.2, monitoring is an essential part of management but not all of it. There are a variety of distributed monitoring frameworks such as Ganglia [48] and Network

Weather Service [49]. The primary purpose of these systems is to provide monitoring of global grid systems and aggregation of metrics collected from various sources.

The Java community has introduced JMX [5] (*Java Management eXtensions*), which enables any Java-based resource to be automatically manageable. JMX technology provides tools for building distributed, Web-based management system for managing and monitoring Java applications, devices and service-driven networks. However, JMX can typically be accessed only by clients using Java technology making it non-interoperable. This issue is being partly addressed by providing a Web service connector for JMX Agents [50]. While JMX presents the capability to instrument applications with appropriate messages, metrics and control mechanism, a Web service based management protocol provides a more cross-platform, standards-based interface.

Similarly, WMI [17] (*Windows Management Instrumentation*) from Microsoft enables local and remote monitoring and management of Microsoft Windows based machines.

A main feature lacking among these management systems is *interoperability*. As previously described, to address interoperability, the distributed systems community has been orienting towards the Web services architecture which is based on a suite of specifications that defines rich functions while allowing services to be composed to meet varied QoS (Quality of Service) requirements. There already are proposals [51] to leverage the Web services management principles in the context of existing management frameworks. The Service Oriented Architecture provides a simple and flexible framework for building sophisticated applications.

2.3.1. Web service based Management Specifications

A crucial application of the Web services architecture is in the area of systems management. WS Management [52] and WS Distributed Management (WSDM) [53] are two competing specifications in the area of management using Web services architecture.

Both specifications focus on providing a Web service model for building system and application management solutions, specifically focusing on resource management. This includes basic capabilities such as creating and deleting resource instances and setting and querying service specific properties and providing an event driven model to connect services based on the publish / subscribe paradigm.

WSDM breaks management in two parts: Management using Web services (MUWS [54]) and Management of Web services (MOWS [55]). MUWS focuses on providing a unifying layer on top of existing management specifications such as CIM from DMTF, SNMP and OMI (Open Management Interface) [56] models. MOWS presents a model where a Web service is itself treated as a manageable resource. Thus, MOWS will serve to provide support for the management framework and support varied activities such as service metering, auditing, SLA management, problem detection and root cause failure analysis, service deployment, performance profiling and life cycle management.

WS Management, on the other hand, attempts to identify a core set of Web service specifications and usage requirements to expose a common set of operations central to all management systems. This minimum functionality includes ability to discover management resources, **`CREATE`**, **`DELETE`**, **`RENAME`**, **`GET`** and **`PUT`** individual management resources such as settings and dynamic values, **`ENUMERATE`** contents of containers and collections, **`SUBSCRIBE`** to events emitted by managed resources and **`EXECUTE`** resource-specific management methods. Thus, the majority of overlapping areas with the WSDM specification are in the MUWS specification. Ref. [57] presents a proposal for evolution of a common management specification.

2.4. Messaging Systems

Distributed systems rely on communication between the resources and the clients. The publish/subscribe paradigm [58] has recently received increasing attention as it is well adapted to support interaction among loosely coupled distributed services in a large scale application. The

primary goal is to glue independent applications together without re-engineering individual components. In a publish/subscribe framework, subscribers register their interest in an event or a pattern of events and are subsequently asynchronously notified of events fired by publishers. Several systems such as Gryphon [59], Siena [60] and Elvin [61] have made significant progress in this area.

Our proposed framework uses NaradaBrokering messaging middleware as the event channel for communicating between various components of the system. It has been developed in the Community Grids Lab at Indiana University [62]. Our choice of using NaradaBrokering was primarily motivated by the fact that it is home grown software with a variety of features such as firewall traversal and support for various security schemes. We now present an overview of the NaradaBrokering messaging system.

2.4.1. NaradaBrokering

NaradaBrokering [63] is a messaging infrastructure, based on the publish/subscribe paradigm, that enables distributed entities to communicate with each other through the exchange of messages. NaradaBrokering has been successfully deployed in the context of collaborative applications, audio/video conferencing applications and GIS systems. Projects that currently leverage the NaradaBrokering projects include the SERVOGrid [64], GlobalMMCS [23], the WEB-IS effort at the Florida State University and the University of Minnesota, and finally, the Anabas [65] system which provides support for shared displays and online collaborative meeting software.

NaradaBrokering incorporates several services such as - reliable delivery [66], ordered delivery, secure delivery of messages [67], access to globally synchronized timestamps, reduction of jitters by preserving time spacing between messages, compression / decompression and fragmentation / de-fragmentation of messages. NaradaBrokering incorporates support for several communication protocols such as TCP, UDP, HTTP, SSL and Parallel TCP: this facilitates

communications in a variety of network realms. The system also supports enterprise messaging standards such as the Java Message Service (JMS) [68]. More recently, the system has incorporated support for SOAP and several Web service specifications [69] such as WS-Eventing, WS-ReliableMessaging [70] and WS-Reliability [71].

2.5. Discussion

The approach discussed in this thesis leverages a distributed messaging infrastructure to provide scalability and a mix of both replication and check-pointing to provide fault-tolerance. Specifically, the architecture implements *passive replication* and *independent check-pointing* which helps provide the desired level of fault-tolerance with simplicity of implementation.

Use of a messaging middleware helps improve the scalability of the system. It provides a simple publish / subscribe interface for delivering and receiving messages. For example, a single entity can easily communicate with multiple other entities by subscribing or publishing to appropriate topics. NaradaBrokering has support for multiple transports and can traverse firewalls, thus allowing communication with entities behind firewalls.

Finally, our choice of management protocol is WS – Management. Our choice of leveraging WS Management was mainly motivated by the simplicity of WS Management and also the ability to leverage WS Eventing [72] implemented recently in the OMII container [73]. We have been using the management architecture for modeling management of a distributed brokering infrastructure [74].

Chapter 3. Management Framework

In this chapter we describe our management framework. The components have been designed keeping in mind the main criteria of management introduced in the previous chapter, namely scalability, fault-tolerance and remote management. We then present an overview of the consistency and security issues that arise in the context of our framework and discuss our approach to address them.

The proposed architecture assumes that external system state required to manage resources is small and can be captured using a message based interface. The external state is defined as the minimum state required in bootstrapping the resource after a failure occurs. The resource's internal state is still managed by the resource. An advantage is that we can use *independent check-pointing* scheme to save resource-specific state at minimal cost. The only requirement is the existence of a scalable, fault-tolerant database which serves as a registry to help store system

state. An example of such a fault tolerant store would be a Context Database [75]. For the purpose of implementation, we implemented a prototype registry that stores all information in memory with an optional extension to store to local file system for fault-tolerance purposes.

Further, the type of resources that require management is large. The number of management interactions required is even larger and specific to the resource in question. There is no *"one shoe that fits all"* manager that can satisfy the management requirements of all possible resources. Hence, we employ a resource-specific manager that encompasses the functionality of managing a specific resource. The management framework provides the basic functionality such as scalability, fault-tolerance, performance and interoperability while resource-specific management can leverage these facilities to provide scalable, fault-tolerant management.

Figure 1 shows a generic management framework. In the context of this thesis, we assume that the *resource to manage* and the *resource manager* are Web services.

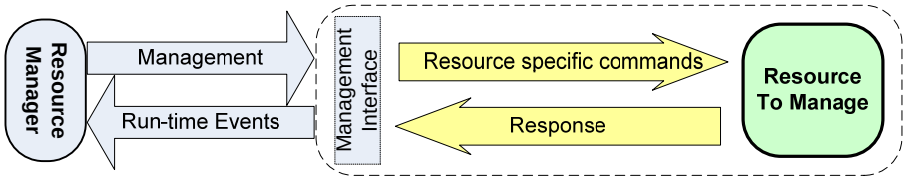


Figure 1 Generic Management Framework

The *Resource* that needs management is any application-specific component. We term such a resource as a *manageable resource*. Usually, with the right configuration, a resource-specific manager can directly interact with the resource and manage it. However, when the resource being managed is not intrinsically a Web-service, a wrapper service that provides a Web-service front-end is required. The *Management Interface* is an entity-specific proxy that has a Web-service interface on one end and an entity-specific interface on the other end. This proxy acts as translator of Web-service based messages to entity-specific command.

3.1. Framework Components

3.1.1. Hierarchical Bootstrap System

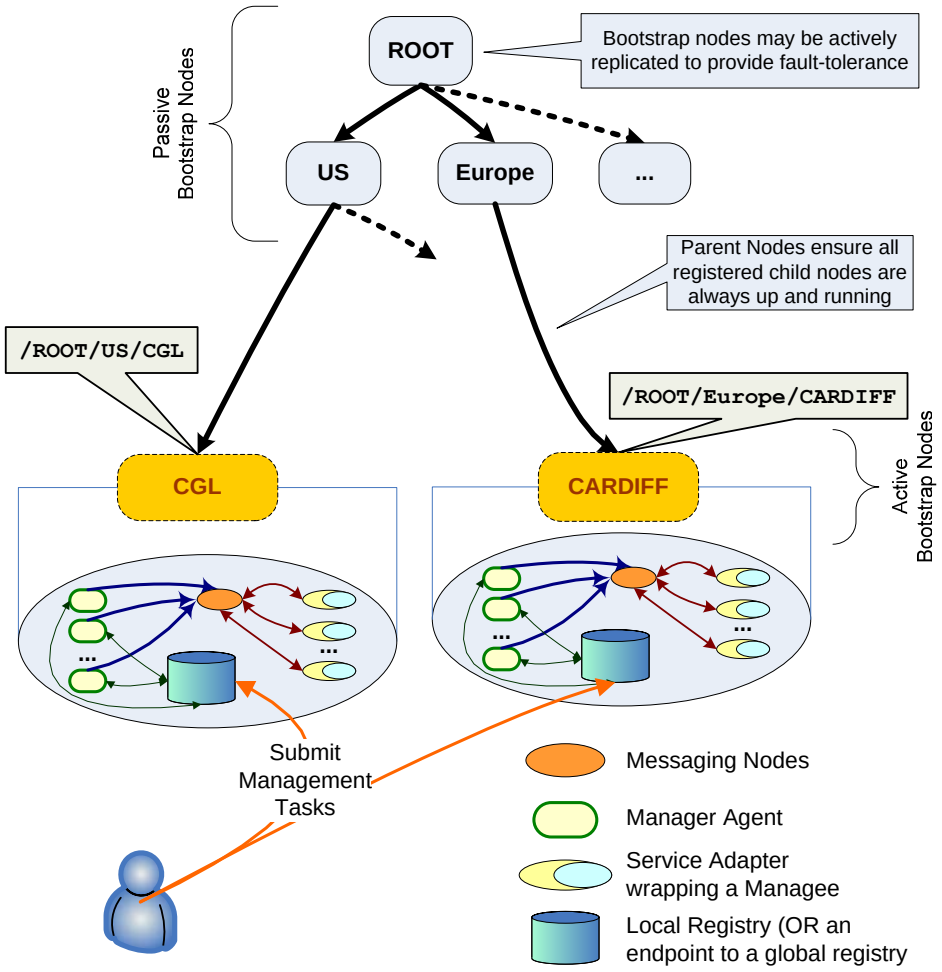


Figure 2 Achieving scalability through hierarchical arrangement

The overall management framework consists of units arranged hierarchically. Each unit is controlled via a bootstrap node. The hierarchical organization of units makes the system scalable in a wide-area deployment.

The bootstrap service mainly exists to serve as a starting point for all components of the system. The bootstrap service also functions as a key fault-prevention component that ensures the

management architecture is always up and running. The service periodically starts, checks the overall system health and if some component has failed, reinstates that component.

The bootstrap services are arranged hierarchically as shown in Figure 2. As shown in the figure, we call the leaf nodes of the bootstrap hierarchy as being active bootstrap nodes. This means that these nodes are responsible for maintaining a working management framework for the specified set of machines (domain).

The non-leaf nodes are passive bootstrap nodes and their only function is to ensure that all registered bootstrap nodes which are their immediate children are always up and running. This is done through periodic heartbeat messages. Failure is detected when a heartbeat is not received within a specified timeframe. We consider the following cases:-

1. If the connection is not lost but the child node has not sent a heartbeat message, the parent node tries to contact the child node and check its health. If the connection can be re-established successfully, the management process continues as before.
2. If the connection is lost, this could be either due to an intermediate network failure or because the actual bootstrap process died.

In either case, the parent node may try to re-establish contact for *K* number of times. If successful, the process continues as usual, else a network alarm is raised (e.g. e-mail the appropriate service provider / administrator with failure details, or the parent bootstrap node automatically re-spawns the child bootstrap node).

In our implementation, we assume that the root bootstrap node is always alive. If this node goes down, the system administrator would bring this up in finite amount of time. Alternatively, the root nodes themselves may be replicated, which ensure that at least one of the nodes is always up and running. For instance, assume a 3-way replication. On failure, the live node can bring up the other two failed node. This follows from the assumption that all three nodes do not simultaneously crash.

We now describe the main components of each unit of the framework. A unit of management framework consists of one or more manageable resources, their associated resource managers, one or more messaging nodes (NaradaBrokering brokers, for scalability) and a scalable, fault-tolerant database which serves as a registry. The arrangement of these components is shown in Figure 3. We now describe each of these components in detail. We will then discuss some of the consistency and security issues in the system and means to address them.

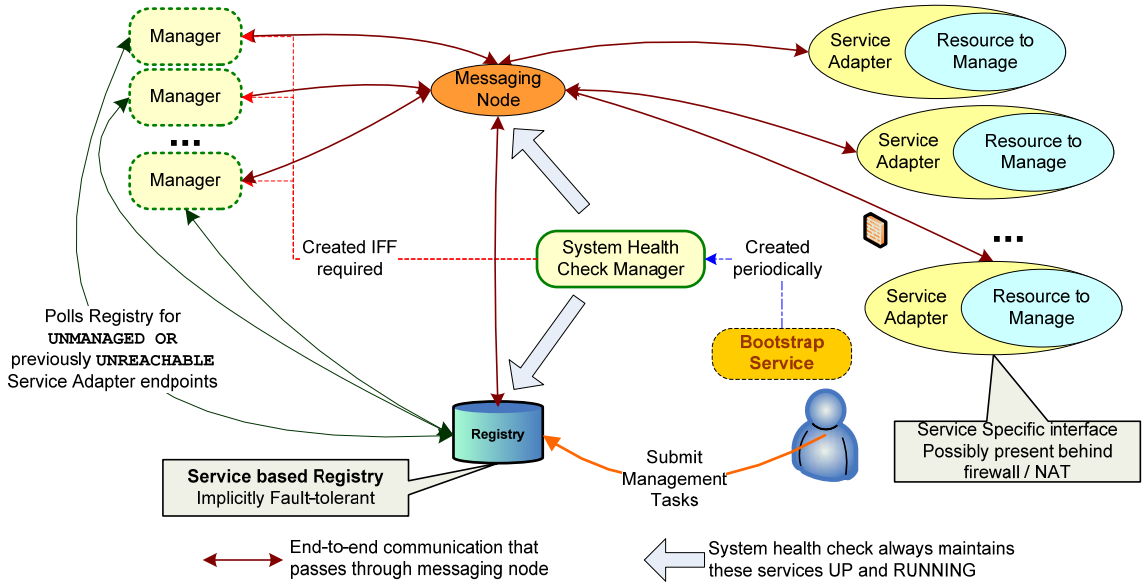


Figure 3 Overview of a Unit of Management Architecture

3.1.2. Managee (Resource to be Managed)

We refer to Managee as the component that requires management. We employ a service-oriented management architecture and hence we expect that these Managees have a Web service port that accepts management related messages. In the case where the Managee is not a Web service, we augment the Managee with a service adapter that serves as a management service proxy. The service adapter is then responsible for exposing the managed resources of the Managee.

3.1.3. Service Adapter

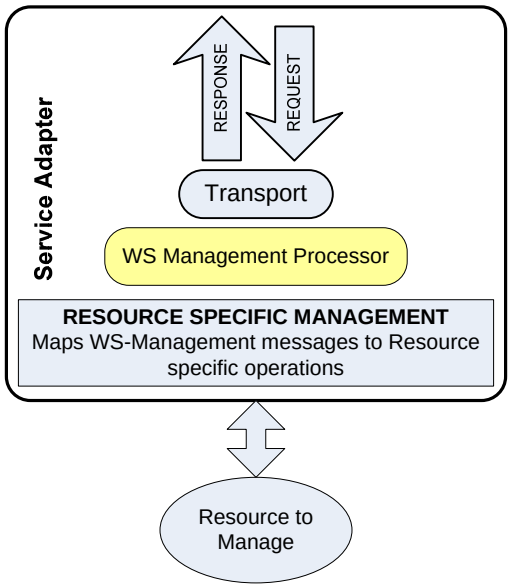


Figure 4 Anatomy of a Service Adapter

The Service Adapter is a proxy that serves to primarily leverage NaradaBrokering's publish / subscribe framework. The Service Adapter also hosts the WS Management processor (Refer Section 4.1) that provides a service-oriented management interface to manage the Managee. The structure is illustrated in Figure 4.

In addition to serving as a management interface, the service adapter also performs the following functions

1. Send regular heartbeat messages to the assigned resource manager process. This helps the resource manager to determine aliveness of the Managee.
2. The Service Adapter may persistently maintain current Managee configuration. Updates to the configuration would be written to stable storage whenever possible. This would allow a failed Managee to be brought up quickly to the last known configuration.
3. Provide transport independent message delivery between the Service Adapter and Manager via Messaging Nodes (Section 3.1.6). QoS is improved by employing multiple

transports. If a particular transport is unavailable (blocked ports, authentication issues), then the Service Adapter tries to poll for another transport that might be usable. E.g. If direct **`TCP`** connection is not possible due to blocked outgoing ports, then messages may be tunneled over **`HTTP/HTTPS/SSL`** connections to the Messaging nodes. Failure of this connection is gracefully handled by retrying several times before concluding failure. Further, the Service Adapter may try different Messaging nodes to connect to, should a particular Messaging node be unreachable after several tries. An alternate way of connecting to the best available messaging node is to use the Broker Discovery Protocol [76].

4. Finally, the service adapter also periodically renews itself with the registry. As will be discussed in later (Ref. Section 3.3.1), every service adapter has an instance id which allows us to track duplicate instances of resources. If the current instance id of service adapter is less than the instance id as seen in the registry, then the service adapter silently cleans up and shuts down. This helps avoid duplicates.

3.1.4. Manager

A manager is a multi threaded process and can manage multiple resources at once. Manager processes typically maintain very little or no state so that they can be easily replaced on failure. This makes the manager robust. Section 5.5 discusses more on the maximum number of requests a single manager can handle. The structure of management process is shown in Figure 5.

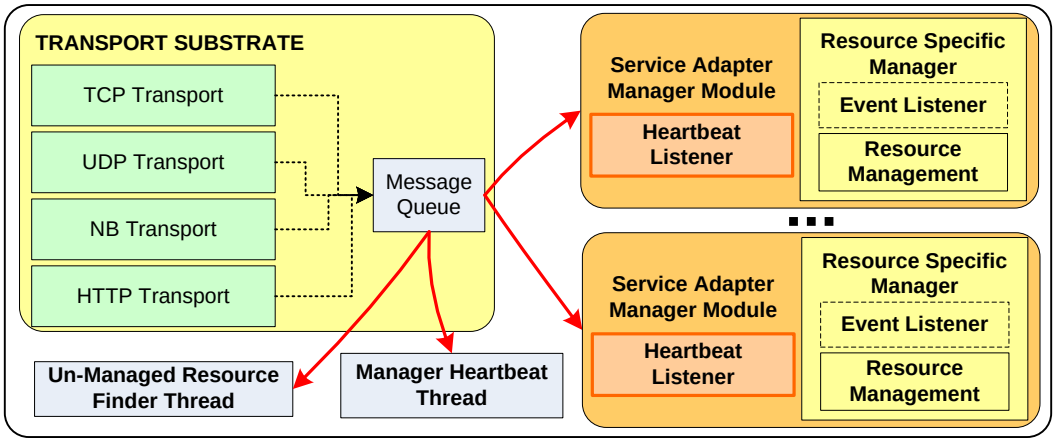


Figure 5 Anatomy of a Manager Process

The Manager process supports multiple transports such as NB or plain UDP / TCP / HTTP. The choice of a particular transport is determined by the quality of service desired.

NB is the most desirable mode of transport, since multiple entities can communicate over NB transport by subscribing to an appropriate topic and publishing to a topic specific to the destination entity. This makes the overall architecture scalable since only one physical connection is required to an available broker for the Manager process to communicate with multiple resources. Further NB can tunnel through firewalls and NAT and thus the Manager can manage resources present behind firewalls.

Using TCP or HTTP is desirable when the target entity supports only these modes of transport. UDP is typically used in communication with the registry rather than leveraging NB, as this makes the Manager – Registry communication impervious to messaging node failures. Further, using UDP also improves scalability of the system as there are no setup and connection maintenance costs. UDP, however, suffers from packet loss and the system must employ retrying of requests that did not elicit a response. Further, some operations may not be idempotent so the system must also detect and discard duplicate requests.

The Manager process starts off with two main threads:-

1. The *Manager Heartbeat Thread* that periodically renews the Manager in the Registry. This allows other Manager processes to check the state of the currently managed resources and if a Manager process has not renewed its existence within a specified time, all resources assigned to the failed Manager are then distributed among other Manager processes.

2. The *Un-Managed Resource Finder Thread* that periodically polls the registry to see if there are any un-managed resources. The number of resources that a particular Manager process manages is determined by the **`MAX_SAM_MODULES`** configuration property. This may be set appropriately by experimentation for the type of resources in question. (Refer Section 5.5 for an analysis of this quantity). If no resource could be found for management, the Manager process exits. This allows the number of managers to automatically adjust to only the required number.

The Manager process also has a queue-based transport substrate. The transport substrate interfaces multiple types of transports as shown in the figure. Messages arriving are placed in an internal Message Queue and are routed to their appropriate destination by a message routing thread. Queuing of messages ensures that at no point is a thread busy processing a message while blocking the incoming messages.

The Manager process spawns off a *Service Adapter Manager Module* for each resource that is assigned to it. This module contains two threads, one specifically for processing heartbeats and another for performing resource-specific management tasks (termed as the *Resource Manager*). If the resource provides events via WS – Eventing, the resource manager may subscribe to these events by maintaining an *Event Listener*.

The message flow between various components is illustrated in Figure 6.

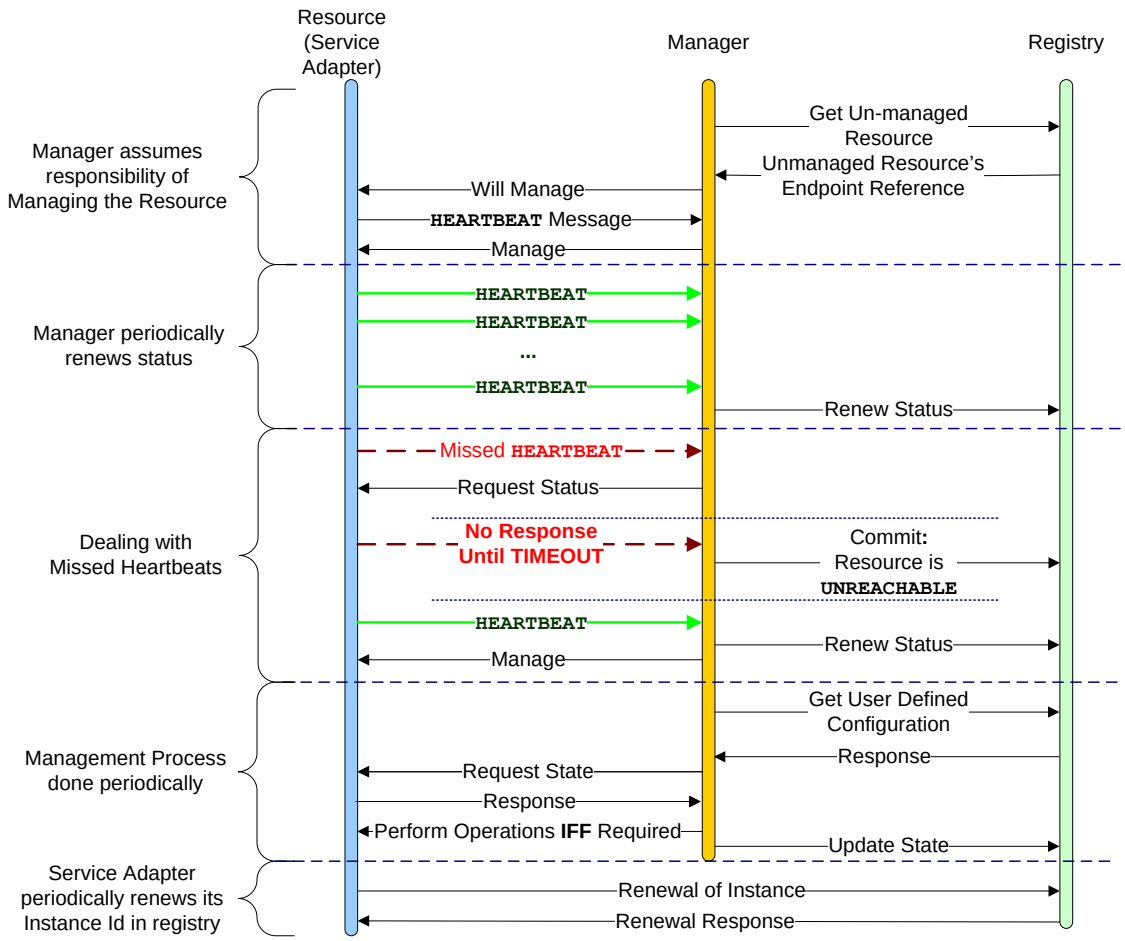


Figure 6 Message flow between components

Web-service Messaging

The manager – managee interaction is based on a Web-service based management protocol. Our current implementation uses WS – Management based interactions. However, WS – Distributed Management could also have been used. WS – Management was primarily chosen because of its simplicity and also to leverage the Web-service based eventing model recently added in NaradaBrokering messaging middleware. Details of the service-oriented messaging framework are provided in Chapter 4.

Managing Service Adapters through Heartbeats

The Manager periodically polls the registry to see if there are any available resources to manage. On finding an un-managed resource, it performs the following steps:

The Manager contacts the Service adapter and registers itself with the Service adapter. This implies that the Service Adapter will send heartbeat events to the registered manager. The manager and Service Adapter may also negotiate communication characteristics such as duration of heartbeat and security.

The Service Adapter periodically sends heartbeat message to the Manager. The manager keeps track of the heartbeat messages to determine the status of the Service Adapter.

If a heartbeat is missed, the Manager tries to contact the Service Adapter.

- If **`SUCCESSFUL`**, the process continues as normal.
- If **`FAILURE`** occurs after several retries, the Manager may conclude the Service Adapter as unreachable and updates the Service Adapter's status in the registry. A consistency problem may occur if two managers try to manage the same resource. Consistency is handled as discussed in Section 3.3.1.

The Manager maintains only short term state and periodically updates the overall state in the registry.

Periodically, the Resource Manager reads a user-defined configuration from the Registry and matches it with the current state of the resource. If the states do not match, the Resource Manager performs operations on the Resource. Finally the state is updated in the Registry.

Managing Managers

The managers follow the *Passive Replication* scheme. If the manager fails for any reason, the manager's state does not get updated in the registry as frequently as it should. When a manager process queries the registry for available resources to manage; the registry also includes resources

whose assigned managers have not renewed their state within the time limits defined by system parameters. Thus, if managers fail, their assigned resources automatically get re-assigned to an existing Manager process. Thus, detection of failure and recovery takes a finite amount of time rather than being instantaneous. An analysis of recovery period is presented in Section 5.8.4.

Fault Tolerance

One would usually run multiple managers. As mentioned previously, manager process maintain very little or no state. This state is regularly committed to the registry. Thus, if a manager process fails then the resource that it was managing can be easily assigned to another manager process. This allows the new manager to continue managing the resource in the event of failure of the old manager process. The newly created resource manager then reads the user-defined state from registry and gathers the current state of resource by querying the appropriate service endpoint. These two states are matched and any inconsistencies are appropriately resolved by the resource manager.

Event Handling

Various application-specific runtime events are generated by managed components (Managees). The resource manager can set appropriate access rights as to which entities are allowed to subscribe to specific events generated by the Managee. In cases where a large number of events are produced or events are produced at a very high frequency, the manager may get overloaded while handling all types of events. This factor should be taken into consideration when designing the system and limiting the number of resources a single manager should manage. Further, in the case that a resource manager does not have the necessary functionality to address the variety of events that might be generated, a separate service responsible for handling specific type of events may be employed. This service may in turn be configured and managed through the management architecture.

3.1.5. Registry

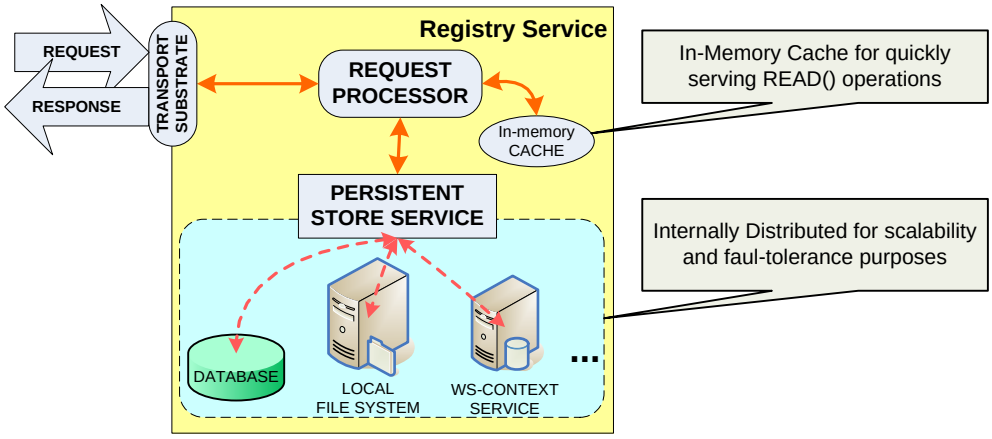


Figure 7 Anatomy of a Registry Service

The Registry stores system state. System state comprises of runtime information such as availability of managers, list of resources and their health status (via periodic heartbeat events) and system policies, if any. General purpose information such as default system configuration may also be maintained in the registry.

Usually read operations can be directly served from an in-memory cache but writes are always written directly to the persistent store. Figure 7 shows an overview of registry internals. The *Request Processor* provides logic for manipulating the data stored in the registry. This mainly includes checking for manager processes that have not renewed within the system defined time frame and to serve as a matching engine to match new resources to managers. The request process service may be replicated to provide fault-tolerance. If case of multiple instances of request processor services, we assume that these services are appropriately configured to load-balance incoming requests while the registry service component as a whole automatically provides consistency of persisted information.

The registry is backed by a *Persistent Store Service* which allows the data written in registry to be written to some form of persistent store. Persistent stores could be as simple as a local file system or a database or an external service such as a WS – Context [77] service. We assume the persistent

store to be distributed and replicated for performance and fault-tolerance purposes. Thus, the local file-system could use a mirrored RAID disk storage for tolerating disk failure. Similarly the database component can replicate at a remote site.

Implementation of such a registry service is out of scope of our current work. As a prototype, we provide an implementation that stores entries in memory and optionally can persist to local file system.

3.1.6. Messaging Nodes

Messaging nodes consist of statically configured NaradaBrokering broker nodes. The Messaging nodes form a scalable message routing substrate to route messages between the Managers and Service Adapters. These nodes provide multiple transport features such as **`TCP`**, **`UDP`**, **`HTTP`** and **`SSL`**. This allows a Managee, present behind a firewall or a NAT router, to be managed (e.g. connecting to the Messaging node and utilizing tunneling over **`HTTP/SSL`** through a firewall).

By employing multiple Messaging nodes, one can achieve fault-tolerance as the failure of the default node automatically causes the system to switch to using a backup Messaging node. We assume that these nodes rarely require a change of configuration. Thus, on failure, these nodes can be restarted automatically using the default static configuration for that node.

3.1.7. User

The user component of the system is the service requestor. A user (system administrator for the resources being managed) specifies the system configuration per Managee which is then appropriately set by a Manager. In some cases, there would be a group of Managees which require collective management. An example of this is the broker network where the overall configuration of the broker network is dependent on the configuration of individual nodes. System configuration (Refer Chapter 6) is set by the user while the execution of necessary actions

is performed by the management architecture in a fault-tolerant manner. The user interface for Broker Management is explained in detail in Appendix A.

3.1.8. Fork Process

For the purposes of our implementation, we assume daemon processes running on hosts where the system is installed. These daemon processes (referred henceforth as "*Fork Process*") are used to spawn various components of the system, such as, bootstrap service, manager processes and whenever possible, even service adapters for certain resources. If a host is physically rebooted, we assume the host's initialization script to automatically start the fork process.

3.2. Summary of components

The various components of our architecture are summarized in Table 1.

Bootstrap Service	Bootstrap service is used to *bootstrap the system* and also ensure fault-tolerance of the entire management architecture. The bootstrap nodes are arranged hierarchically to scale the system over a wide area. This is detailed in Section 3.1.1.
Managee	Managee represents the actual resource being managed. Aliveness of these components is detected by periodic heartbeat events sent by the associated service adapter. This is described in detail in Section 3.1.2.
Service Adapter	Service adapter serves as a mediator between the manager and the Managee. The service adapter component is explained in more detail in Section 3.1.3.
Manager	Manages the Managee by sending appropriate messages to the service adapter. The interactions are based on a Web-service based management protocol such as WS Management, although WS – Distributed Management could also be used. The

	Manager is explained in detail in Section 3.1.4.
Registry	The registry component is used to maintain system state. Details are provided in Section 3.1.5.
Messaging nodes	Messaging nodes provide a scalable, transport protocol independent messaging substrate. Details are provided in Section 3.1.6.
User	The user is the service requestor and functions as the administrator of the resources being managed. This is explained in detail in Section 3.1.7.
Fork Process	Used to spawn processes on remote hosts. Explained in detail in Section 3.1.8.

Table 1 Summary of Architecture Components

We now discuss how the desired features mentioned in Section 1.3 can be addressed:

Fault-tolerance is primarily achieved by leveraging a fault-tolerant registry to store systems state. The system components are themselves made fault-tolerant by periodically running *system health-checks* to ensure that all crucial components of the management architecture are alive. A hierarchical bootstrapping mechanism ensures that individual domains are alive and on failure, can try to instantiate the failed domains.

Scalability is achieved on two levels: (a) In each individual unit of management framework, the use of a messaging substrate improves scalability by reducing the number of physical connections required between a manager process and the messaging node and (b) The units themselves are arranged hierarchically to scale the system for wide-area deployment.

Interoperability is achieved by leveraging a Web-service based messaging protocol for management of resources.

The system can deal with any type of resource (hardware or software) by suitably wrapping the resource with a wrapper to provide a Web service interface. This makes the framework generic for use with any type of resource.

Finally, the system is autonomous and handles failures with minimum user interaction. This makes the system usable for management needs.

We will discuss performance in Chapter 5.

3.3. Issues in Distributed System

A distributed system has to deal with various issues to successfully function. We have addressed scalability and fault-tolerance as two important characteristics. We now discuss two other issues namely, *Consistency* and *Security* and provide the means to address them.

3.3.1. Consistency

Distributed systems are faced with many consistency issues such as duplicate requests, messages arriving out of order and multiple instances of resources leading to inconsistency. Specifically, this raises a number of consistency issues in our system such as:

1. Two or more managers managing the same resource;
2. Old messages reaching after newer messages;
3. Multiple copies of resources existing at same time (Orphaned resources). This is true when resources are software services and the management framework spawns a new instance of the service to account for a possible failure of the old instance and

4. Multiple system health check routines spawned by the bootstrap service may see incoherent system state exacerbating the consistency of the system state.

To address these issues we impose the following consistency check scheme:

i. We rely on the registry to generate a unique Instance ID (IID) per instance of resource or manager thread created. This ID could be an NTP timestamp or a simple sequence number that is guaranteed to be unique and monotonically increasing. Further, we assume that when the registry generates this number, all replicas of the registry have the same view of the number.

ii. Every time a new resource requires management, it needs to register itself with the registry. Further, each service adapter periodically registers its presence in the registry. This facility is required to determine duplicate resources (in the case when the management framework can create new instances of resources such as by means of spawning processes). The resource's service adapter automatically gets its instance id when it registers in the registry and is returned via the registration response. Also, during renewals, the registry simply returns the current known instance id.

iii. A resource-specific manager thread also obtains its unique *Instance ID* when it is assigned a resource to manage. This unique id is used by the resource-specific manager to construct a unique *Message ID* to be used for every message sent from that entity. This *Message ID* is a combination of the sender's *Instance ID* and a monotonically increasing sequence number. Retries of requests use the same *Message ID* rather than generating a new *Message ID*. This allows the resource to discard duplicates.

We now discuss, how the above inconsistencies may be resolved using these restrictions:

1. If a manager process is considered dead / unreachable due to a missed / delayed heartbeat, the health check spawns a new manager process to take over the responsibility of managing the resources which were being managed by the previous manager. If the

old manager tries to invoke a management operation on the resource, the service adapter looks at the message and can disregard the old manager's request. Thus, a request coming from a manager with Instance ID A (IID_A) is considered by the resource's service adapter to be obsolete if the resource is currently being managed by a manager with Instance ID (IID_B) and if $IID_A < IID_B$.

2. By keeping track of the last known successfully processed message's message ID, duplicates and obsolete messages may be discovered and discarded.

3. In some cases, a user-defined policy may cause new instances of resources to be spawned when an old resource is deemed unreachable. In such cases, if the old resource comes back up, we get multiple duplicate resources existing at the same time. This may lead to application specific inconsistencies.

   We assume that in such a case, if a user defined policy states that a new resource can be instantiated, a user defined policy also exists on how to deal with multiple copies of resource. Given this assumption, duplicate resources may be detected and inconsistencies may be resolved as follows:

   a. Consider a resource-specific managers M and an instance R_1 of resource R. Assume, M is managing R_1 and at some point concludes that R_1 is unreachable. M then instantiates R_2 by virtue of a user-defined policy.

   b. Soon after, R_1 comes back up and sends heartbeat / event to manager M. In the current implementation, M discards this heartbeat / event. Hence, it will no longer manage R_1.

   c. Further, R_1 periodically renews itself in registry also. The registry response is simply the current known instance id of the resource R. When R_1 sees that R_2 exists such that $R_2 > R_1$, R_1 silently shuts down.

4. To prevent multiple health checks from running at the same time and introducing inconsistencies in the system, we implement the following policy:

    a. Health check always runs for a pre-determined time interval x, during which it either reports success back to the bootstrap node OR self-terminates, if it cannot successfully bring up failed components of the management framework. Further, it also sends periodic heartbeats to the bootstrap service to note that it is alive and running.

    b. If the health check routine becomes unreachable from the bootstrap service and is unable to deliver heartbeats in timely fashion, the bootstrap node may conclude that the health check routine is dead and spawn a new health check process. This may introduce race conditions and inconsistencies if the old health check process is not actually dead.

    c. To prevent this, the bootstrap service will always wait a little over the time interval of health check *x*, before re-spawning a new health check routine, unless the existing health check routine has reported success. Thus, even if the previous health check is unable to deliver heartbeats, it would have self-terminated after its interval *x*.

As a final note, the auto instantiation scheme presented above poses a problem when systems get partitioned and managers in each partition spawn duplicate processes to compensate for all missing processes. As an illustration, if one partition (A) contains about 97% of resources, while the other partition (B) contains 3% of resources, the managers can end up building an extra 97% of missing resources on partition B. This problem would become significantly complex if there were 3 partitions containing 98%, 1% and 1% of resources. While we consider this situation as being out of scope of our current work, we imagine specification of appropriate mechanisms and policies to handle such situations.

As an illustration, a failure-recovery mechanism could state that spawning processes to compensate for missing resources should be done only if the missing resources do not exceed $y\%$. A user-defined policy would set $y = 3$, thus ensuring that managers do not spawn more than a fixed number of duplicate resources.

3.3.2. Security

A distributed system gives rise to several security issues such as but not limited to:

1. Denial of service attacks
2. Unauthorized users accessing resource
3. Man in the middle attack that impersonates an entity
4. Malicious users modifying messages (such as when a messages passes over insecure intermediaries)

The NaradaBrokering messaging substrate provides schemes for dealing with such types of issues. We present an overview of these schemes below:

1. The *Topic Creation and Discovery scheme* [78] ensures that the physical location of the entities (such as host and port) is never revealed. This is the first step towards providing denial of service. Further, the topic creation scheme ensures that every entity has a unique topic and discovery of this topic is restricted in many ways such as by providing an access control and presenting valid credentials. This prevents unauthorized clients from finding resources or sending garbage messages to essential services.
2. The *Security Framework* [67] provides secure end-to-end message delivery of messages. The security infrastructure requires all entities to provide valid credentials before they can exchange messages. Entities are identified by the entity's digital certificate which is validated against a root certificate authority's certificate. Once validated, the entity can request creation of a session key which can be used for all further communication with

that entity. This scheme provides security against man-in-the-middle attacks, message modification and inspection of encrypted messages when traveling over insecure intermediaries.

3. Use of digital signatures helps establish trust and detect modification of message by malicious entities.

Chapter 4. Service-Oriented Management

In this chapter, we present an overview of the implementation of WS – Management framework for modeling management interactions. As described in Section 2.3.1, the WS Management framework only defines the minimum required interactions and the application is free to extend beyond this minimum specification. Further, a manageable endpoint is not required to support all interactions specified (such as **`GET`**, **`PUT`**, **`CREATE`**, **`DELETE`**, **`RENAME`**) but only those that make sense in the particular context of the application. Further, not all manageable resources would provide enumeration or the eventing model. However, it is required that if an application intends to support these models then it must implement the WS Enumeration [79] and WS Eventing [72] specifications, respectively.

4.1. The WS Management Processor

The WS Management specification provides basic functionality by leveraging WS-Enumeration [79], WS – Transfer [80] and WS – Eventing [72] specifications. Any other resource-specific operations can be defined in addition, if the existing operations are not sufficient for the management requirements. The basic WS Management processor can then be deployed using the Service Adapter to provide a WS – Management interface. The WS Management implementation contains components as shown in Figure 8.

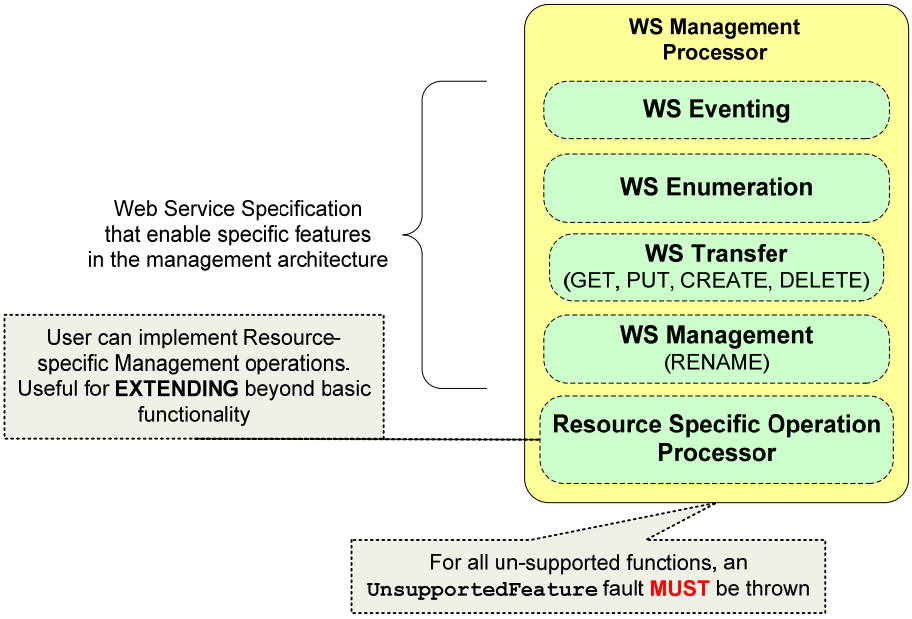


Figure 8 WS Management Processor

Once a message is received, the WS Management processor processes the message according to the steps illustrated in Figure 9. The processing begins by checking the *maximum envelope size* for the response. WS Management specifies that if the maximum envelope size is specified then the minimum size must be 8192 octets to reliably encode all possible faults. This element may be discarded if **`mustUnderstand`** is set to **`FALSE`**. When **`TRUE`** and the value is less than 8192, a fault is thrown.

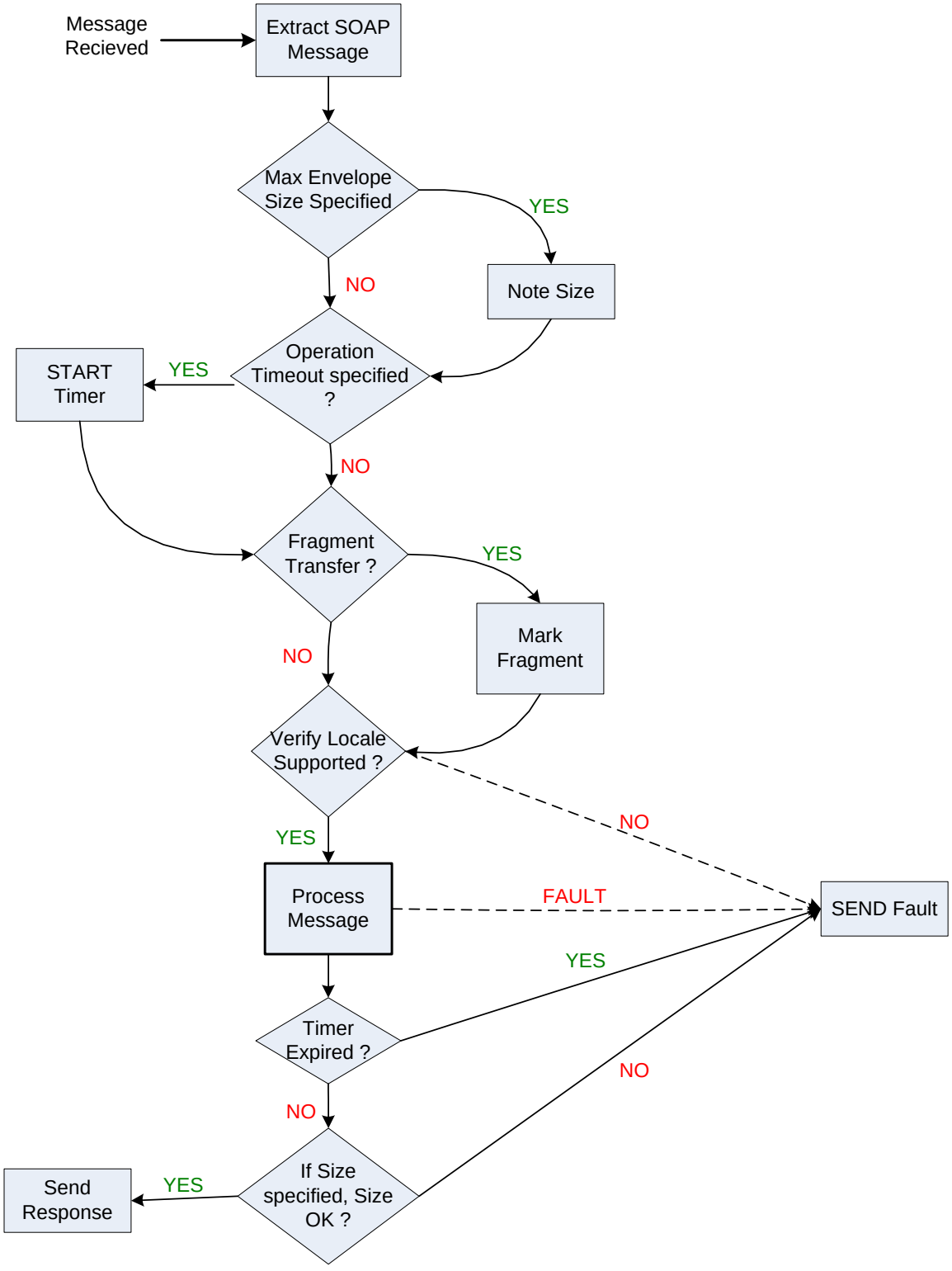


Figure 9 WS Management message processing flowchart

Management operations typically span locales, and many items in responses can require translation. Typically this applies to descriptive information intended for human readers which is sent back in the response. Translation of such information to a specific language can be specified by an optional locale element. If the message processor is unable to process the requested translation, an appropriate fault is thrown. In our implementation, non-English locale results in a fault.

An operation timeout may be specified to indicate that a response is desired within the specified timeframe. If specified, a timer is started. If a response is indeed generated (success or failure) the timer is cancelled. However, if the timer expires before the processing has finished, a **`TimedOut`** fault is sent back to the requestor. WS Management states that any state changes that may have occurred during the processing, are later inspected by the requestor by making one or more **`GET`** requests to retrieve the service state.

Other processing such as Fragment Transfer and Locale check are done. Fragment transfer is required when the request size is too big to send in one single message hence the request may be fragmented. It is the service's responsibility to determine the logic of dealing with fragmented requests. Finally, the **`Process Message`** block is invoked. Depending on the characteristics of a particular resource, only a subset of operations may be supported.

A final check happens just before the message response is sent. As discussed above, if the maximum response size constraint is not met, a fault is sent instead of the actual message. According to WS Management, the resource itself may be queried for state which would provide a hint as to the success or failure of previous operation. Idempotent operations may be re-invoked by specifying a suitable maximum response size.

4.2. WS Transfer

The main operations (**GET, PUT, CREATE, DELETE**) are leveraged from the WS Transfer specification and this forms the basis of WS Management processor. The framework provides an abstract class **WSManProcessor** which must be extended to provide the correct functionality. The **WSManProcessor** provides abstract functions for the above operations including the **RENAME** operation (defined by WS Management). It is the service writer's responsibility to return an **UnsupportedFeature** fault when the resource does not wish to provide functionality for one or more operations.

If the resource provides runtime events or allows for enumeration, the functionality can be provided by extending the **WSEvProcessor** or **WSEnProcessor** respectively. Further, for extensibility purposes, where the resource requires enabling management through a richer set of functions, the **ResourceSpecificOperationProcessor** maybe extended to provide resource-specific functionality.

We now describe each of these functionalities below. An example of how a resource management framework may be built is shown in Appendix D.

4.3. Eventing

WS Eventing allows a management interface to send notifications to interested entities (referred to as event subscribers). We leverage WS Eventing from NaradaBrokering's support for Web service Eventing. NaradaBrokering provides the event generator (event source) and event receiver (event sink) interfaces. Our architecture reverses these roles. Thus, the **WSEvProcessor** (Service Adapter side) behaves as an Event Sink for the generated events. These events are sent over the Service Adapter's transport over to the Resource Manager where a **WSEvClient** functions as an Event Source and provides events to the appropriate sink running on the Resource Manager. This is illustrated in Figure 10. A separate component (the WS Eventing

Subscription Manager) must be run. This component interacts with the Event Generator to store valid subscriptions. This component is itself a Web service and is currently accessible by sending messages to the **`SUBSCRIPTION-MANAGER`** topic.

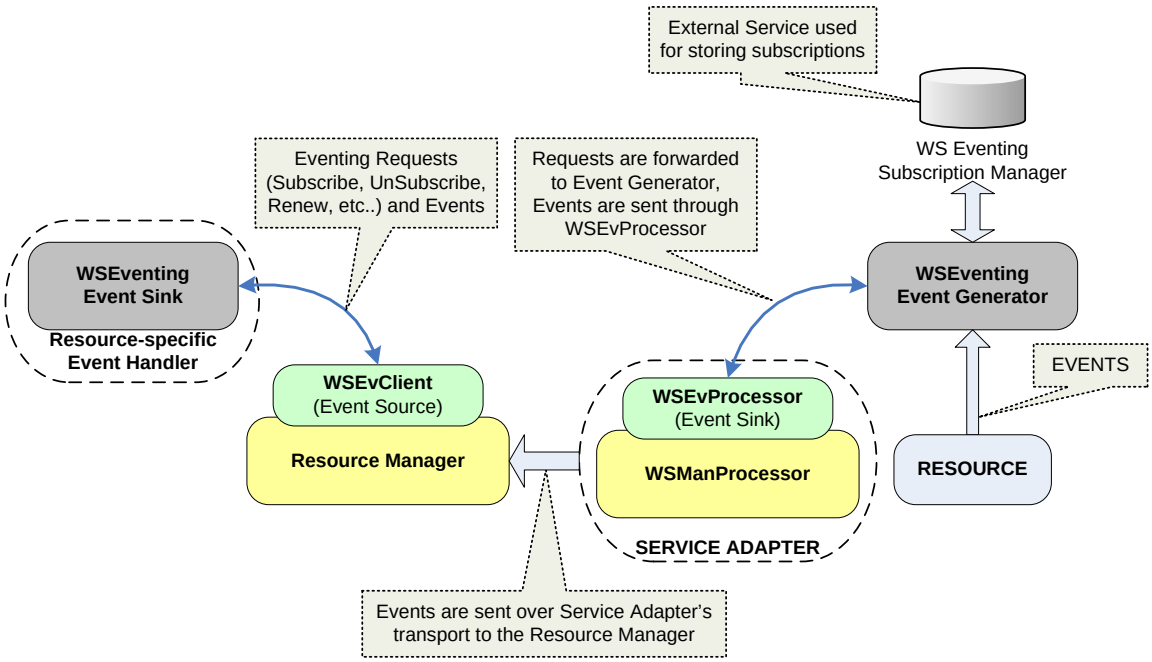


Figure 10 Event Flow between Resource and Resource Manager

4.4. Enumeration

Usually managed resources produce logs or contain multiple entries in internal containers which form the resource state. Such entries may be retrieved by enumerating the resource's containers. The resource may provide such a capability by implementing the **`WSEnProcessor`**.

The WS Enumeration processor internally assigns unique ids for enumerations (referred to as *Enumeration Context*) for each valid enumeration. Invalid enumeration contexts (e.g. expired or unknown) are automatically dealt with by the framework during validation of incoming enumeration requests. However, the responsibility of actual data management lies with the resource. Further, the resource can control supported operations. Thus, for example if the

resource does not support enumeration renewals then the service adapter must throw an appropriate fault such as **`UnableToRenew`**.

4.5. Extensibility

Resources may provide additional resource-specific functionality which cannot be modeled by the default operations provided. This functionality is provided by implementing the **`ResourceSpecificOperationProcessor`** abstract class. This interface provides a simple function **`processResourceSpecificMgmtRequest`** which can define additional resource-specific management message handling.

4.6. Summary

In this chapter, we presented a brief overview of the two Web service based management protocols. We discussed the reasons for our choice of implementing WS Management and provided an overview of the WS Management framework.

Chapter 5. Performance Analysis

In this chapter, we present an analysis of the system and present benchmark results. The main purpose of benchmarking analysis is to show the feasibility of the system. We describe our benchmarking approach and include observed measurements.

5.1. Introduction

Our first experiment is to establish a base level for the maximum publish rates supported by a NaradaBrokering Broker. To measure this, we setup a measuring subscriber that sums up the total messages received in a 5 second interval. Our observations indicate that the broker can support in excess of **`5000 messages / sec`** when the message size is about 512 bytes and in excess of **`4500 messages / sec`** when the message size is about 1024 bytes.

Since majority of the message interactions comprise of messages which can be encoded using ~ 1 Kbytes (Refer Table 3, Table 4), we assume "**`5000 messages/sec`**" as the maximum publish rate that can be supported by the broker. We use this as the basis for all the analysis presented henceforth.

5.1.1. System Configuration

All our experiments were conducted on the Community Grids Lab's *GridFarm cluster* (GF1 – GF8). Table 2 summarizes the configuration of the machines

Component	**Details**
Processor	Dual Intel Xeon (2.4 GHz) Hyper threaded CPUs.
Memory	2 GB RAM
Operating System	Linux 2.4.22-1.2199.nptlsmp
Java Version	Java Hotspot™ Client VM (build 1.4.2_03-b02, mixed mode)
Network	Linked via a 1 Gbps link

Table 2 Test Machine Configuration

5.2. XML processing overhead

The WS Management framework (presented in Chapter 4) was implemented in Java. We have used the XMLBeans [81] version 2.0.0 for mapping the schemas to Java Objects. WS Management relies heavily on the SOAP 1.2 specification, specifically for handling faults. SAAJ (Soap Attachments API for Java) version 1.3 supports SOAP 1.2. However, to maintain compatibility with other leveraged software packages, we implemented our own SOAP marshalling and un-marshalling framework using XMLBeans.

XMLBeans does not validate documents by default. Hence, before each requested operation is executed, we validate the corresponding XML against its schema. In some cases, additional processing is necessary for completely inspecting the message. This processing primarily includes

matching the requested **<wsman:ResourceURI>** and **<wsman:Selector>** with the resource instance.

In this section, we present some results on the costs associated with marshalling and un-marshalling the various management messages in the system. The benchmarks are presented in the context of messages used in managing the NaradaBrokering messaging system (Refer Chapter 6).

We measure the time taken to create and process a request and response message. Table 3 shows the timings and typical message sizes for requests. The *marshalling* time refers to the time required to create the request. This message is then processed by a **WSManProcessor** which checks for various WS Management constructs such as *OperationTimeout, Locale, and MaximumEnvelopeSize* and also is responsible for conversion of byte representation of XML to usable Java objects. After this step, the message (Java representation) is forwarded to the actual resource's management interface which may do additional processing on the message. Specifically, this step checks for valid input data.

Once a response is determined, we measure the time it takes to create the response. Table 4 lists the associated response messages and typical message sizes. In the *marshalling* step the response is checked for maximum envelope size and then marshaled. This response message is then processed by the **WSManClient** which *preprocesses* the message to check for faults before delivering it to the resource manager thread. The Broker specific XML processing refers to inspecting the response and validating the XML.

Operation	Typical Message Size (Bytes)	Marshalling time (msec)	Unmarshalling (msec)	
			Generic XML processing	Broker Specific XML Processing
Create Broker	813	1.332	2.411	0.078
Get Broker Information	809	0.928	1.377	0.059

Create Link	1194	1.189	1.36	0.703
Delete Link	1102	1.207	1.558	0.626
Delete Broker	934	0.954	1.381	0.236
Set Broker Configuration	2590	4.028	1.428	1.301

Table 3 Time spent in processing WS Management formatted REQUEST messages for Broker Management

Operation	**Typical Message Size (Bytes)**	**Marshalling time (msec)**	**Unmarshalling (msec)**	
			Generic XML processing	**Broker Specific XML Processing**
Create Broker	1108	0.565	1.058	0.274
Get Broker Information	1398	0.454	1.26	0.528
Create Link	1214	0.602	1.027	0.301
Delete Link	726	0.363	0.936	0.161
Delete Broker	726	0.353	0.911	0.154
Set Broker Configuration	724	0.381	0.87	0.217

Table 4 Time spent in processing WS Management formatted RESPONSE messages for Broker Management

Each of the above operations was run 1000 times and the average time was reported.

To find the average XML overhead per operation we add the marshalling time for XML, the generic and broker specific XML processing for requests and their associated responses. Thus, for example, the "*Create Broker*" operation takes about

$$1.332 + 2.411 + 0.078 + 0.565 + 1.058 + 0.274 \approx 5.7 \text{ msec}$$

We observe that, the average total XML overhead is about 6 msec per operation. The "Set Broker Configuration" request operation puts each broker property and hence the message size is much

higher. However, this operation is invoked only once before creating a broker instance and hence the overhead is acceptable.

For responses, the **DeleteBroker** and **DeleteLink**, operations take approximately the same time since in both cases a **<ResourceDeleted>** element is sent back confirming the deletion.

5.3. Maximum Resources managed per Manager Process

In our architecture, the Manager process is multithreaded. Per resource being managed, there are 3 threads: the first periodically checks the heartbeat status for aliveness of managed resource, the second monitors the events originating from the Managee, and the third performs the actual management work. A separate set of threads maintain internal queuing whose primary work is to dispatch incoming messages to appropriate threads for further processing. Finally, transport level objects require their own set of threads (e.g. a TCP Listener thread that listens for incoming TCP connections / UDP listener thread that listens for incoming datagram packets). We maintain a connection pool with the messaging node so there would be additional P threads which represent the size of connection pool (P). Thus, per Manager process there are a total of

$$D*(TH_{Management} + TH_{HeartbeatCheck} + TH_{EventProcessing}) + P * TH_{Transport} + TH_{ManagerProcessing}$$

$$= D*(1 + 1 + 1) + P * 1 + TH_{ManagerProcessing}$$

$$= 3D + P + TH_{ManagerProcessing}$$

Where, **D** is the number of resources a manager process is managing and P is the size of connection pool with the broker (typically ~ 10). Further, $TH_{ManagerProcessing}$ is a very small number (~ 5). Thus, we get, number of threads

$$\approx 3D + 15$$

Further, each resource has to maintain some resource-specific state that gets updated and written to registry at regular intervals. This data is very resource-specific and determines the number of

threads that can be spawned. Figure 11 shows the number of threads that can be spawned when each thread maintains a specific amount of resource state.

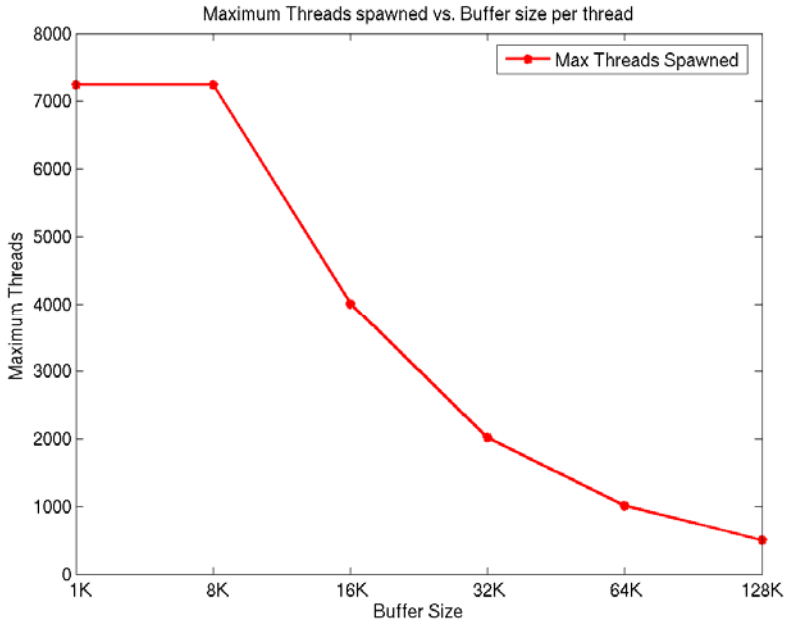


Figure 11 Maximum threads spawned when each thread maintains resource state of specified buffer size

The state size per resource is very dynamic and is very resource-specific. We present the analysis in case of broker management in Section 6.9.1.

As an illustration, assume that the state size for a typical resource is 16 Kbytes. Note that not all threads in a system maintain 16K state. Assuming only the management threads maintain 16K buffers and heartbeat threads maintain much less state, the total possible threads would be approximately $(7000 + 4000) / 2 \approx 5500$.

With $3D + 15$ threads per process and approximately 5500 allowed threads, we get $D \approx 1800$. Thus, *theoretically,* approximately 1800 resources can be managed by a single Manager process. However, as the number of threads increases, the response time per thread also increases (Refer

Section 5.5). System policy dictates how much response delay each resource can tolerate which would affect the number 1800 derived above.

5.4. Initialization Costs

The steps in the initialization process are as follows:-

```
Repeat while More Resources can be Managed
{
      RSA = Get UNMANAGED Service Adapter from Registry
      Initialize Service Adapter Manager Module
            for Registered Service Adapter (RSA)
}
```

As discussed above, a single manager process can manage multiple resources (theoretically ~ 1800) but can only process a specific number of concurrent requests. We analyze this factor in more detail in Section 5.6. The initialization cost typically involves reading from registry and initializing the resource –specific management thread. Read from registry typically takes about 5 msec and depends on the amount of data read from registry. Initialization of resource-specific management thread is a resource-specific activity. We note that this is a one time cost.

5.4.1. Discussion

The registry is the central part of the whole system and limits the number of queries that can be executed every second. The "*Get* **UNMANAGED** *Service Adapter*" operation is not a simple "*Get*" operation but involves going through all existing Registered Service Adapters and finding the first best match.

This usually involves finding a service adapter which was previously deemed **UNREACHABLE** or the service adapter was never **MANAGED** or if a **MANAGED** service adapter is found then the

associated manager has not successfully renewed in the **MAX_ALLOWED** interval. The currently implemented algorithm is summarized in the following pseudo-code:

```
FOR EACH Service Adapter x REPEAT
IF x.status == UNREACHABLE, then that SAM-Module has relinquished
control of the service adapter. Now find another service adapter 'y'
such that
      y = ServiceAdapter
      WHERE ((y.status == UNREACHABLE ||
                  y.status == REGISTERED AND x != y)
            ||(y.status == MANAGED &&
                  (currentTime - y.manager.lastRenewal > MAX_ALLOWED)))


IF (y != NULL)  then such a service adapter is indeed found
            set y.status = MANAGED;
            set y.manager = myManagerID
            return y;
```

Here, the first condition selects a new service adapter *y*, if the following 2 conditions are met

1. *y* was previously unreachable or *y* has recently REGISTERED and was never managed.
2. *y* is currently being managed but the Manager process currently managing *y* has not renewed itself in the maximum allowed time.

The second condition above, automatically assigns management of resources to alternate managers when their associated managers have failed.

For simplicity, the current implementation implements this as an $\mathbf{O(N^2)}$ operation. To see how, note that the first request goes through only 1 record, the second request goes through 2 records

and so on. The N^{th} request goes through all N records. Thus, on an average **1 + 2 +...+ N = N*(N+1)/2 = O(N^2)** reads are done.

This may be reduced to an **O(N)** operation at the cost of maintaining 2 tables, one for **UNREACHABLE** and **REGISTERED** service adapters and another one for **MANAGED** service adapters. Only when a suitable service adapter is not found in the first table, the second table will be consulted. Once a service adapter is selected, it would be moved to the **MANAGED** service adapters table. Similarly, when a service adapter is deemed **UNREACHABLE**, it is moved to the first table. Lookup in first table is **O(1)** since always the first record would match. Lookup in second table will still be **O(N^2)**, but would help to reduce startup costs. If there is any specific resource matching to be performed, this step would still require going through the entire table to find the best possible match.

5.5. Runtime Response Costs

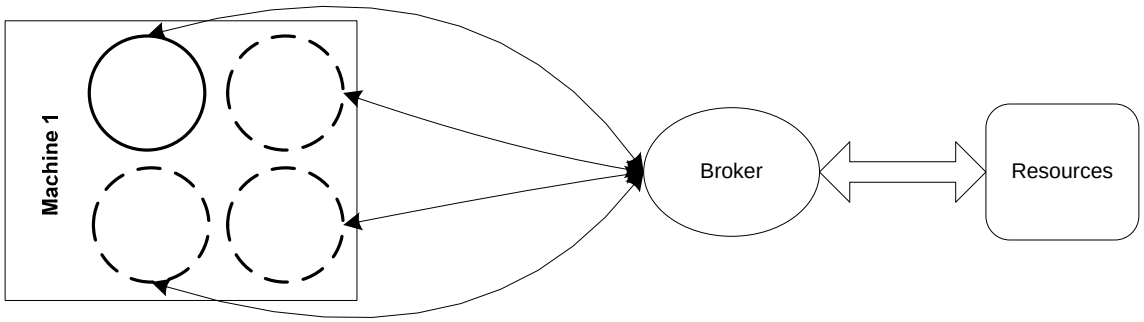
The most important deciding factor which determines the maximum number of requests a manager process can handle is how the response time varies as the number of resources being managed by a single process increases. This would also enable us to formulate the number of Manager processes required and the number of resources that can be managed by a single instance of the management architecture. We define a single instance as comprising of one or more messaging nodes, one Registry and one or more Manager processes. Finally, this number also determines how the system scales.

The test setups are shown in Figure 12. As shown in the figure, we increased managers on single machine (Setup A) and multiple machines (Setup B). The testing methodology was as follows:

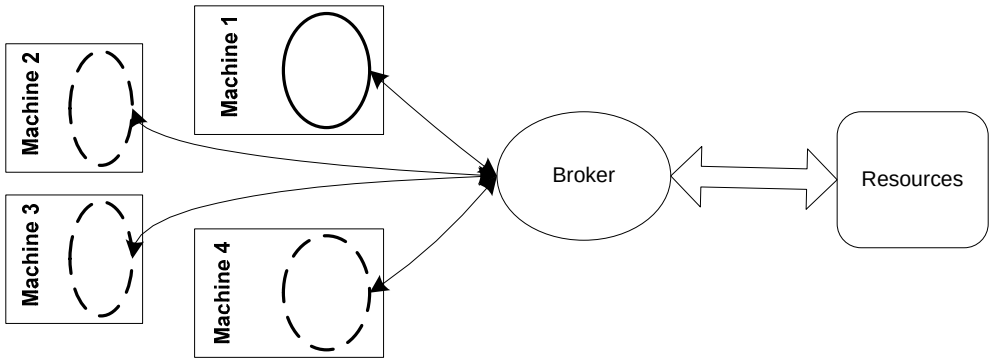
1. The resources are run via a thread pool that sends pre-generated events to the managers.
2. Just before sending these events, we start a timer.

3. In response to the events, the resource-specific manager thread responds back to the resource.

4. When all managees get their corresponding response, we stop the timer and the difference corresponds to the overall response time.



Setup A: Running Managers on same machine



Setup B: Running Managers on multiple machines

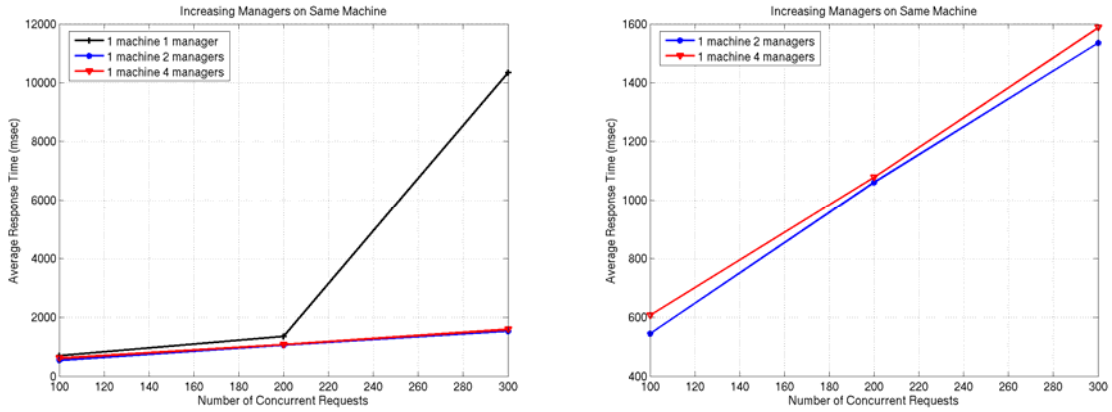
Figure 12 Test Setup

5.5.1. Observations

As expected, with an increase in the number of managed resources, the number of threads per manager process grows (roughly 2 x Number of Managed resources). Thus, the average response time increases. In our case, there was no registry access during processing of the event, however, this behavior is resource-specific and may require one or more registry accesses in certain cases.

The detailed results are included in Appendix B. We summarize the observations below:

1. Figure 13 (a) shows the average response time as we increase the number of managers on a single machine. Figure 13 (b) shows a close-up of the average response time for 2 and 4 managers. We note that on the same machine, increasing managers beyond the number of available processors does not improve average response time. This is because the processes contend for processors and the hence the observed time for running 4 managers on a 2 processor machine is slightly higher than running 2 manager processes on a 2 processor machine. We attribute the difference to context switching between processes.

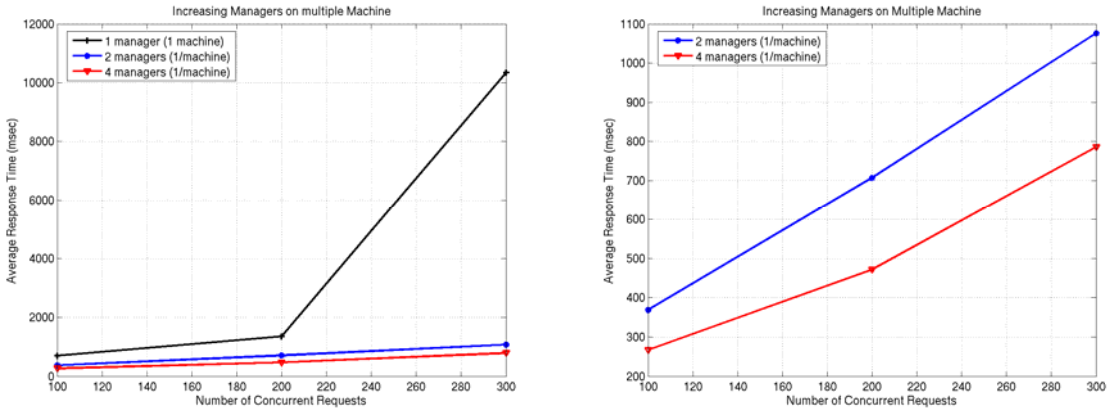


(A) (B)

Figure 13 Increasing managers on same machine improves performance. However, there is no significant difference when number or manager processes is greater than number of processors on the machine

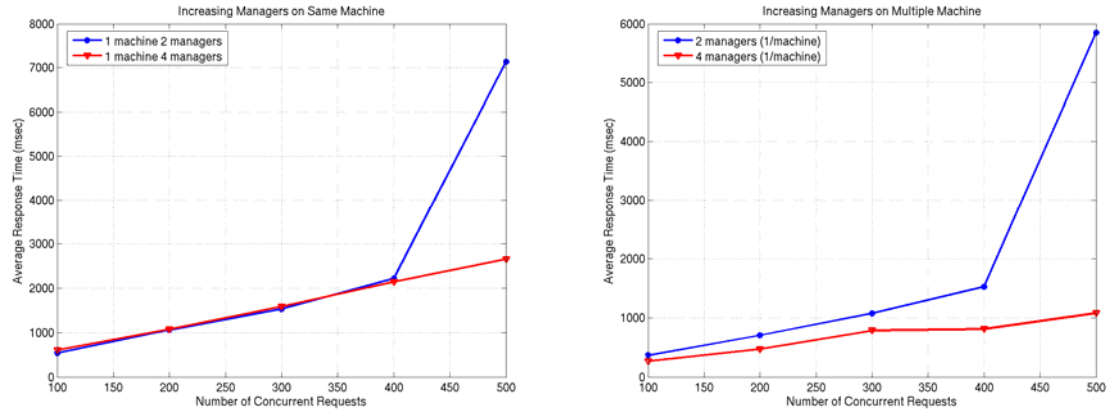
2. Further, running managers on different machines decreases the average response time as number of managers are increased (Figure 14).

3. Finally, we note (Ref. Figure 15) that average response time increases linearly with the number of concurrent requests but shoots up as the managers become saturated.



(A) (B)

Figure 14 Increasing managers on multiple machine improves performance



(A) (B)

Figure 15 Managers saturate and response time stops increasing linearly

5.6. Performance Model

Figure 16 shows the components of the average response time. These components are as follows:-

T^P = Processing time per request at the manager

T^X = Message transit time at broker. This value is ≈ 1 msec when broker is not saturated

T^R = Processing time at Resource (which is minimal for testing purposes) and represents the un-marshalling cost of response. Our test measurement does not consider cost of processing the response, but simply the minimal cost associated with arrival of response (un-marshalling XML).

Apart from these factors, there is network latency L^{MB} between Manager and Broker, and L^{BR} between Broker and Resource

Thus, for any event, the total event handling time T_E is the sum of all above:

$$T_E = T^P + 2 * (L^{MB} + T^X + L^{BR}) + T^R \quad \text{.....................................} (1)$$

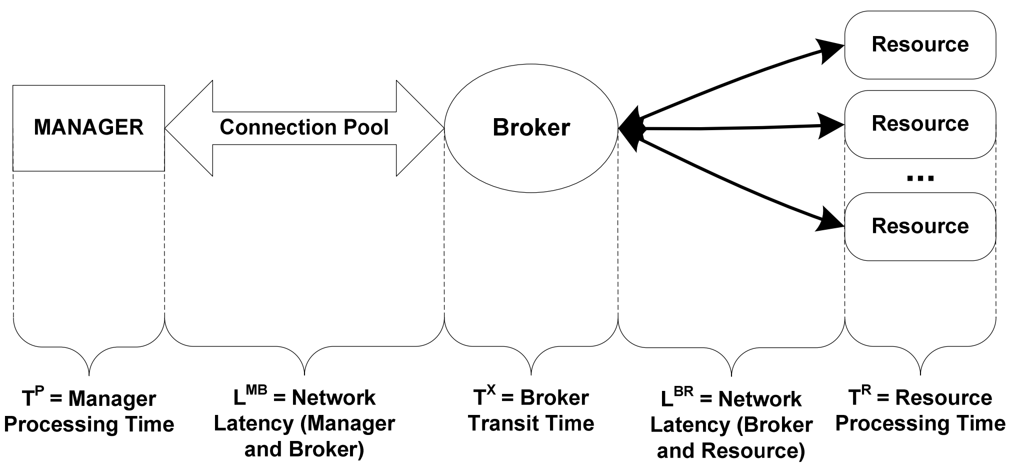


Figure 16 Modeling components of response time as seen by the resources

Note that the multiplier 2 refers to request and response. We now discuss the various factors as follows:

To compute T^X, note that a single broker (when not saturated, can give throughput up to 5000 messages /sec) for message sizes 512 bytes, and > 4500 messages/sec for message size 1024 bytes.

For sake of illustration, let us assume the maximum throughput to be about 4000 messages /sec.

For N concurrent requests, we have N responses and so as long as **2N < 4000**, the broker is not saturated, i.e. the broker transit time can be ignored for N < 2000. Since we posit, that each broker can support up to a maximum of 800 resources we consider T^X to be very small.

Further the Network latency will be considered as a constant L i.e.

$$L = 2*(L^{MB} + L^{BR}) \quad \text{.....(2)}$$

Further, T^R is constant and represents the un-marshalling costs of the response (about 1 msec). We consider all constants as

$$K = L + 2*T^X + T^R \quad \text{.....(3)}$$

Thus, the total event handling time (as seen by individual resources), from (1), (2) and (3) is:

$$T_E = T^P + K \quad \text{.....(4)}$$

Further, T^P can be broken down into time required for the processing thread to perform additional operations ($T^{EXTERNAL}$), pure processing ($T^{CPU}_{MANAGER}$) and an additional time spent in process scheduling ($T^{SCHEDULING}$).

The external operations include one or more registry or disk accesses. If external calls are blocking calls, this allows other requests to be handled while the thread blocks on response from external dependent components. In our experimental setup, there is no external dependency while handling the event and so $T^{EXTERNAL} = 0$. So our model refers to a case where the only time spent is processing the message and there is no dependence on external factors.

$T^{SCHEDULING}$ becomes significant when there are more processes than available processors. For E.g.: This explains why the average response time when running 4 managers on single machine is slightly greater than running 2 managers on a single machine, where the machine, GridFarm, has 2 available processors per machine. While, further analysis of this factor is out of scope of our current work, we note that this factor should be considered when determining the number of managers that must be run per machine given a desired quality of service (such as average response time when all managed resources simultaneously generate events).

Finally, $T^{CPU}_{MANAGER}$ is a Resource-specific activity that includes the necessary processing including un-marshalling of request and marshalling of corresponding response.

Thus, we get,

$$T^{P} = T^{CPU}_{MANAGER} + T^{EXTERNAL} + T^{SCHEDULING}$$

Further, on hyper-threaded processors, multiple requests can be processed simultaneously. If **C** is the number of threads that can be simultaneously active, then up to **C** requests can be processed in time T^{P}. Thus, the average time required for processing **C** requests is T^{P}/C and the total time for processing N requests (T^{PROC}) is

$$T^{PROC} = (N/C)*T^{P} \quad \ldots\ldots (5)$$

N = Number of concurrent requests

T^{P} = Processing time per request on manager's side

C = Maximum number of threads that can execute simultaneously (**C = 2** in our case, for hyper threaded processors)

As an illustration, we collected the average of event processing times for 150 resources (using a single manager on 1 machine) and we get an average value of T^{P} = 8.37 msec. Thus,

$$T^{PROC} = (N/C)*T^{P} = (N/2)*8.37 \approx 4.2*N \quad \ldots\ldots (6)$$

Thus, total observed time (theoretical, assuming T^{P} = 8.37) for processing N concurrent events is

$$T^{OBV} = T^{PROC} + K$$

$$= (N/C)*T^{P} + K$$

$$\approx 4.2*N + K \ldots\ldots (7)$$

Here K represents the constant (that combines network latency, broker transit time and un-marshalling time on resource). Since the number of resources only affects the processing time at manager, this constant is independent of N as long as the broker is not saturated with processing messages.

Further, our test setup runs on Grid farm machines which has every processor hyper-threaded i.e. it can run a max of 2 threads simultaneously **(C = 2)**. Thus, the maximum request processing rate by a single multithreaded manager process is

$$\mathbf{D = (C/T^P)\ requests/sec}$$

Further, the manager will not be overloaded as long as the total requests to be processed are <= maximum outgoing rate i.e. <= **D**. When the manager is managing more than **D** requests, the manager gets overloaded. Hence, we would see degradation in performance. Thus, **D** determines the maximum number of concurrent requests that a single manager can handle with linear increase in response time.

As an illustration, theoretically (for T^P = 8.37 msec and C = 2),

$$\mathbf{D = (C/T^P) * 1000 = (2/8.37) * 1000 \approx 239\ requests\ /\ sec}$$

To find the observed break point of manager we steadily increase the number of concurrent requests on 1 manager. The test setup puts the value of **D ≈ 210**, as observed in Figure 17.

Figure 17 Saturation point for a single manager process

Thus, we conclude that a single manager can be assigned (**D ≈ 200**) resources, subject to the conditions of test setup such as resource-specific event handling. While **D** could be much higher than 200, weighing other factors such as desired average response time, we limit the value of D to 200. For sake of illustration, we use the value of **D = 200** in future calculations when determining percentage of additional management infrastructure required.

5.7. Amount of Management Infrastructure Required

We now try to answer the research question, *"How much Management Infrastructure is required to handle N Resources?"*

We make one assumption as the basis of our analysis. As discussed in previous section, a single manager process can handle a maximum of D requests with a linear increase in response time. We assume that a single manager is assigned no more than D resources so that we get the average response time as a linear function of D. Further, this also puts a lower bound on the number of components required.

If only the Manager – Managee interaction is carried over the messaging substrate then the messaging substrate needs to support **N + (N/D)** simultaneous connections. To improve performance, we usually have a connection pool between manager and broker. Let **P** be the size of this connection pool. Similarly, an additional connection pool may be used for the registry. Let **M** be the maximum number of connections allowed by the messaging node. Then we have

N + (N/D)*P + P <= M

or N <= (M-P)D/(P + D)

Thus, for example, in our measurements, the broker could reliably support about 800 simultaneous TCP connections. As an illustration, if **D = 200** and **P = 10**, then the total resources per domain are approximately 752. Various approaches may be used to scale to support larger number of resources. One way is to switch to a less costly transport protocol such

as UDP. UDP is connection-less and hence there is no limitation on the maximum open file descriptors. However, it is also unreliable. Hence, more logic must be employed to deal with missed, duplicate or out-of-order messages. The second approach is to use a strongly connected cluster of brokers. This approach requires additional management in setting up links between the various messaging nodes and maintaining them in a fault-tolerant fashion. A third way would be to redistribute resources such that they are in different management domains, possibly creating additional child domains as necessary.

Note that the main node that limits the number of connections is the messaging node (broker). However, a broker is not absolutely required unless a subset of resources is behind firewalls / NAT devices. Further, using a broker also implies that a manager need not maintain a separate connection for each resource it manages, which is required when using HTTP/TCP. Assuming a broker is required; the amount of infrastructure required can be computed as follows:

1. Let **M** be the maximum number of resources that a single messaging node can support. Thus, to manage **N** resources, we require **CEILING (N/M)** messaging nodes.
2. Assuming 1 messaging node per leaf domain, we require **N/M** leaf domains (Refer Section 3.1.1).
3. Further, per domain we need at least **M/D** manager processes per leaf domain. A single leaf domain would also have its own boot strap node and one or more instances of registry (or registry endpoint).
4. Finally, let **R** be the minimum number of registry instances required. Thus, if we use a 3-way replicated registry database, we have **R = 3**.

Thus, total number of management infrastructure processes at the lowest leaf level is

(Components/Domain) * Number of Domains

= (R registry + 1 messaging node + 1 bootstrap node + M/D managers) * (N/M)

= (2 + R + M/D)*N/M

To manage the **N/M** leaf domains, an additional number of passive bootstrap nodes are required. Typically the number of passive nodes would be **<< N/M** and we ignore it for the purpose of this analysis.

Thus, for managing **N** resources we require an additional **(2 + R + M/D)*N/M** processes. Thus, we compute the number of additional resources required as the percentage of additional infrastructure with respect to the number of resources being managed as follows:

$$MGMT_{INFRASTRUCTURE}$$

$$= \frac{[(2 + R + M/D)*N/M] * 100}{N} \%$$

$$= [(2+R)/M + 1/D] * 100 \%$$

To see how much is this value and how it is affected by different factors, we will consider a few cases. In all the numerical calculation presented below, the assumption is that the value of N is very large. If N were small (e.g. N = 10), we still require the basic management framework components such as messaging node, bootstrap service, 1 or more manager processes and registry.

5.7.1. Using 4-way replicated registry and typical values for D and M

Assuming typical values of **D** and **M** are **D = 200** and **M = 800**. When each leaf domain has its own registry and assuming its 4-way replicated, we have **R = 4**. Then,

$$MGMT_{Infrastructure}$$

$$= [(2+R)/M + 1/D] * 100 \%$$

$$= [(2+4)/800 + 1/200] * 100\ \%$$

$$\approx 1.2\%$$

5.7.2. Using a shared registry

In the case where a common registry is used, each leaf domain would have 1 registry service interface that is always maintained up and running. Thus, $R = 1$ and we get,

$$MGMT_{Infrastructure}$$

$$= [(2+R)/M + 1/D] * 100\ \%$$

$$= [(2+1)/800 + 1/200] * 100\ \%$$

$$\approx 0.87\%$$

5.7.3. If a messaging node is not used

If a messaging node is not used, then the managers have to open a separate connection to the resource it is managing. This poses a problem in cases where the resources being managed are behind a firewall and the firewall blocks incoming connections. In case where there are no firewalls, then the percentage of management infrastructure can be computed as follows:

$$MGMT_{Infrastructure}$$

$$= (R \text{ registry} + 1 \text{ bootstrap node} + N/D \text{ managers})$$

$$= 1 + R + N/D$$

Thus, percentage of management infrastructure is

$$= \frac{1 + R + N/D}{N} * 100\%$$

$$= [(1+R)/N + 1/D] * 100\ \%$$

For very large N $(N >> R)$, the percentage of infrastructure is

$\approx (1/D) * 100\ \%$

Assuming $D = 200$, we get

$MGMT_{Infrastructure} = 0.5\%$

5.7.4. Varying the number of maximum resources managed by a single Manager

Let us keep the maximum number of entities that can be handled by a single messaging node (**M**) to 800. This is a number that typically represents the maximum ports that a single process can successfully open[3]. This number will also vary in some cases, depending on the maximum message/sec the messaging node has to support.

Figure 18 shows how the percentage of additional management infrastructure varies with **D**. We see that even when we fix the maximum clients per messaging node to 800 and have a shared registry (**R = 1**), the additional management infrastructure required is < 5% until **D = 25**. Reducing the value of D below 25, exponentially increases the additional infrastructure required. Finally, when for every resource there is a single manager process more than additional infrastructure required is more than **100%**. The reason why this value is slightly more than 100% is because, apart from managers, there are additional components such as registry, messaging node and bootstrap service that make up the management infrastructure.

[3] Typical configuration allows any single process to open a maximum of 1024 files (files, sockets, devices) although this may be changed through appropriate system configuration.

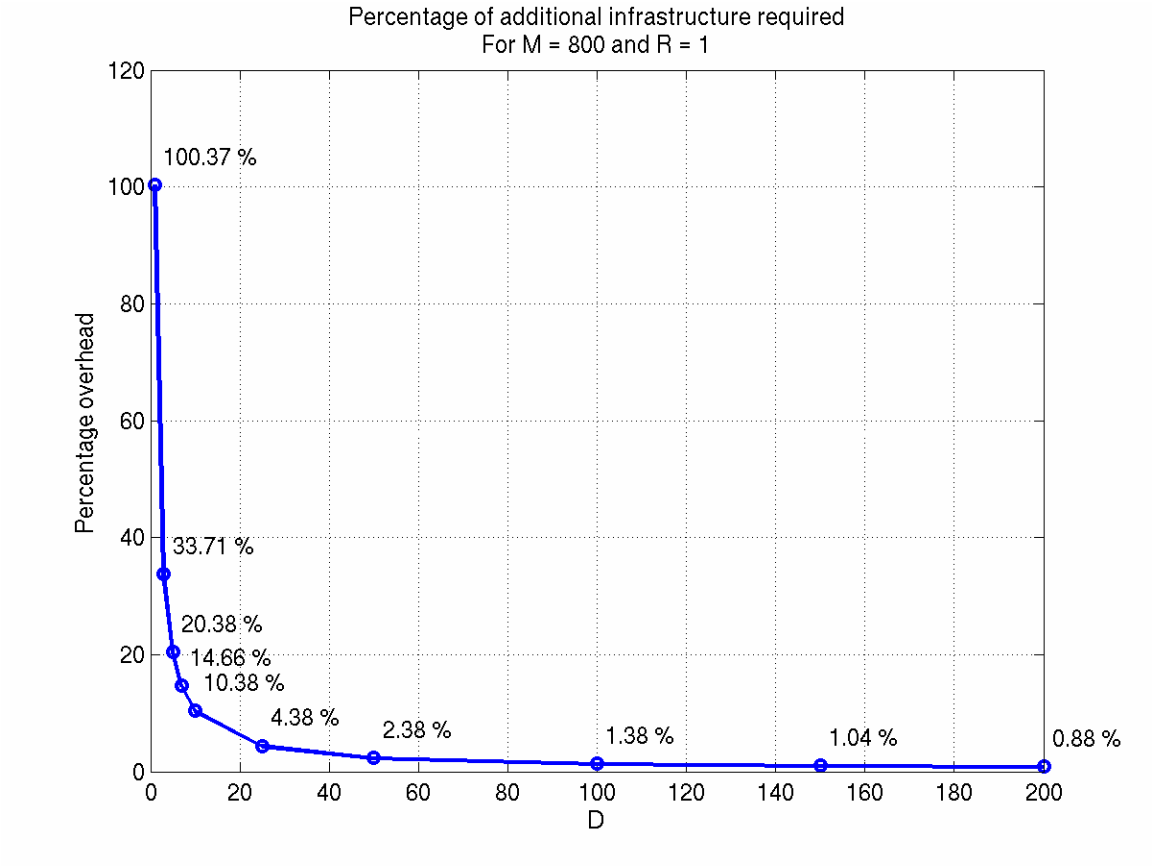


Figure 18 How Additional Infrastructure varies with number of resources a single manager can manage

Thus, we conclude that for large N, and when `D ≈ 200`, fault-tolerant management of the system can be achieved by adding about `1%` more resources. Thus, the approach uses an acceptable number of extra resources.

5.8. Failure Recovery Costs

5.8.1. Resource Failure

The system follows the following algorithm to recover from failure:

1. Check heartbeat interval
    a. If `lastRenewal > 5 * MAX_HEARTBEAT_INTERVAL`

THEN check if I'm still the registered manager of the resource in question.

- The **MAX_HEARTBEAT_INTERVAL** is set to **10 sec**, so the difference is usually **50 seconds**

This step usually involves a READ operation from Registry.

b. If I'm not the currently registered owner (my heartbeat reached late and it was concluded that I failed) then relinquish control of the resource.

2. If I'm indeed the registered manager, then proceed to re-register myself with the resource's service adapter.

a. If the service adapter has not responded after **MAX_RETRIES**, the correction procedure kicks in. The correction procedure is usually resource dependent and could be to simply notify the user or actually try to re-instantiate another resource instance if possible.

- The **MAX_RETRIES** is set to 2 with a **10 sec** wait period for each try, thus the resource manager waits for another **20 sec** before concluding failure.

Thus, the total time to start the recovery phase is

$$T_{Detection}$$

$$= 5 * MAX\_HEARTBEAT\_INTERVAL + MAX\_RETRIES * WAIT\_FOR\_RESPONSE$$

$$\approx 70 \text{ sec}$$

After this period, the actual correction process begins. Recovery time is a resource dependent quantity and thus we define the total recovery time as

$$T_{Recovery} = T_{Detection} + T_{Correction}$$

A discussion of the correction time for Broker Management is discussed in Section 6.9.

5.8.2. Registry Failure

During the initial startup or after a failure, a registry has to reload all data from the persistent store service. Alternatively, data may be loaded only when requested. There are trade-offs when using either of the approaches; improving registry startup cost vs. worsening time to serve read requests. The current implementation simply loads all entries from persistent store. The time to reload the registry from context depends on the number of resources and the data stored per resource. Thus, there are 1 or more resource-specific elements. Further, for each registered components (Service Adapter, Manager, Messaging Node), there is 1 metadata object. To put this in perspective of the total objects, we define the following

$N_{ResourceElements}$ = the number of Resource-specific entries per resource

$N_{ServiceAdapters}$ = the total number of service adapters.

D = number of resources assigned to a single Manager process

Then, $N_{Managers} = N_{ServiceAdapters}/D$

$N_{MessagingNodes}$ = number of messaging nodes used for communication. This is usually very less (1 – 5) and may be ignored.

Thus, the total number of registered objects in the Registry

$N_{Objects}$

$= N_{ServiceAdapters} * N_{ResourceElements} + 1 * N_{ServiceAdapters} + 1 * N_{Managers} + 1 * N_{MessagingNodes}$

Assuming that the registry allows 200 operations per second, the time required to retrieve / store all objects is $N_{Objects}/200$ seconds.

As an illustration, if

$N_{ResourceElements} = 5$, $N_{ServiceAdapters} = 100$, $D = 10$ and $N_{MessagingNodes} = 1$,

then the total time

$T_{Retrieve}$

$= (5*100 + 1*100 + 100/10 + 1) / 200$

$= 611/200$

≈ 3 seconds.

Thus, on failure, for the system to regain its last state, about 3 seconds overhead would be spent in reloading the entire system state from persistent store in addition to other initialization costs (e.g. starting up the registry process and establishing appropriate connections).

5.8.3. Messaging Node Failure

Failure of messaging node causes loss of connection between all Managees and their associated Managers. The bootstrap service periodically (e.g., every 10 seconds) runs a health check manager that detects failure of the messaging node by sending PING requests to the messaging node and checking for responses. If a messaging node is down, a messaging node is forked off by sending a message to the appropriate Fork Daemon process. The Service Adapter and Manager processes detect failure of connection to broker and automatically try to re-establish contact with the broker. Once a connection is re-established, management process starts off where it left.

Since the health check manager runs only periodically (e.g. once every 10 seconds or more and is configurable), system failures can only be detected within 10 seconds. Thus, the system may be un-responsive for about $10 + T_{Restore}$ time period, where $T_{Restore}$ is the time required by the system to restore to the state before failure. This typically should only entail operations that cause the Resource Manager to read state from the Managee (Resource) it is managing.

5.8.4. Manager Failure

When a Manager process fails, management of all the associated Managees fail. As the number of resources managed by a manager process increase, the number of Managees which loose a

Manager also increases and so does the recovery time. Further, Manager failure is detected relatively late. As an illustration, the usual heartbeat interval for Managers is $H_{Manager} = 5$ **seconds**. A Manager is considered failed for all practical purposes, if it fails to renew for **4 *** $H_{Manager} = 20$ seconds. After this interval, all resources managed by the failed Manager are automatically assigned to a new or an existing manager process, which has fewer than D resources assigned.

Again, after a Manager process failure, all assigned Resources become **UNMANAGED**. After 20 seconds (this is a system dependent parameter and may be adjusted as required), the system health check process will detect a failed Manager and invoke another Manager process to take care of the **UNMANAGED** resources. If **D** is the number of resources a manager process manages, then the Manager process needs to load the information for **D** Managees. If **K** is the average number of information items that comprise the state, then the total resource-specific information that needs to be loaded from the registry is **D*K** and the total message exchange is **2*D*K**. Further this is a read-only operation and can be easily served from an in-memory cache. Assuming about **5 msec** retrieval time (includes processing + latency) per information object, the total time is about **5*D*K** msec. As an illustration, if **D = 100** and **K = 10**, then time required will be **≈ 5 sec**. Additional time is spent in informing the Service Adapters of the new Manager after which the usual process of management continues.

5.9. Discussion

A set of benchmarks in context of NaradaBrokering management were presented and in all cases, the average XML processing overhead was found to be **≈ 5 msec**. The total overhead for the *"Get Broker Information"* operation is about 4.6 msec. This operation represents a periodic health check routine invoked by the manager process to check the aliveness of broker and validate runtime configuration. All other operations would be used primarily during initialization and during recovery from failure and thus their costs are acceptable.

From the observations presented in the chapter so far, we note that a single manager process can handle about 200 requests simultaneously. This number is *typical* while the *actual* number is dependent on the resource being managed and time varies with various other interactions such as one or more registry accesses. We assume that this factor would be appropriately adjusted to achieve the desired quality of service.

Every registry access typically adds **`5 msec`** overhead. However, as the number of entities doing a registry access increases, requests are queued as the registry is unable to keep up with the deluge of requests. Thus, ideally, there should be no frequent registry accesses to minimize response time. This implies that the runtime state maintained by the resource manager must be sufficiently small as to be updated in registry by as few calls as possible.

Thus, the management architecture scales better when the resources can be managed by maintaining only small amount of runtime state.

Chapter 6. Prototype and its Evaluation

Messaging based distributed brokering infrastructures have gained much popularity in recent years in the distributed computing community. They have been instrumental in helping to provide clear demarcation between the application logic and Quality of Service aspects. These brokering systems employ a large number of connected peers called brokers which form a messaging substrate. To get the maximum benefit from the services provided by the messaging substrate, it is required to setup these brokers and connect them in topologies specific to the application. To demonstrate the use of the management architecture, we have implemented management of a grid messaging middleware (NaradaBrokering).

6.1. Motivating Example

Various topologies [82] on connecting these peers exists, each based on differing routing, fault-tolerance and cost characteristics. Run-time metrics are gathered using monitoring techniques

[83] which measure various aspects of the system that enable us to understand the performance of the system and in some cases, provide hints on improving the performance. This naturally leads to re-deployment of the brokering network with a different configuration. To summarize, we need an architecture that enables us to rapidly bring-up and tear down a broker network. It is also required to set specific configuration settings for every broker and have the ability to change the configuration on-the-fly. We term these actions collectively, as management of the brokering infrastructure.

Other peer-2-peer (P2P) based systems use static topologies which may be inefficient in some cases. P2P systems based on distributed hash tables such as Chord [84] use a bootstrap node to get a node address. Future additions automatically get address from one or more previously initialized nodes when they join the network. CAN [85] uses a similar approach where an incoming node contacts a bootstrap node to retrieve a set of randomly chosen nodes. These systems do not take network distances in to account when creating the routing table which may result in certain lookups resulting in overlay hops spanning the entire diameter of the network.

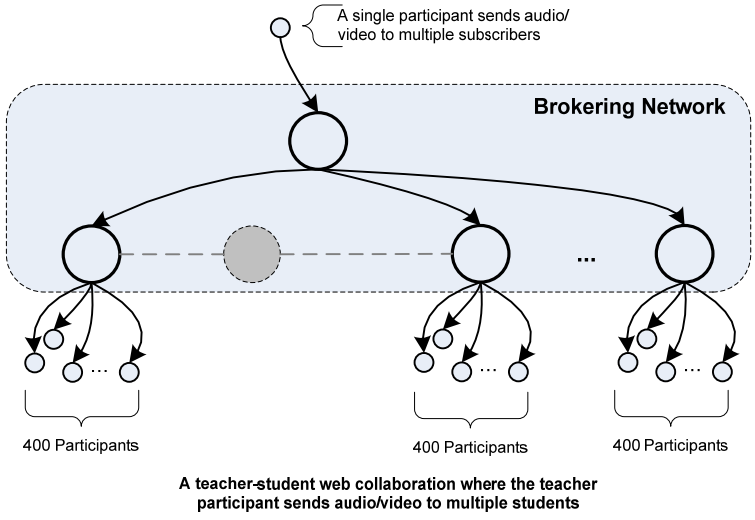


Figure 19 Teacher - student relationship based collaborative session

Consider the problem of deploying a brokering network for supporting 10000 clients in a collaborative [23] fashion. Ref. [86] shows that a single broker can support up to 1500

simultaneous participants with audio streams with very good quality audio while about 400 participants can simultaneously receive video with acceptable quality. For a higher number of participants, we can employ a tree-based structure as illustrated in Figure 19. The problem lies in deploying the brokering topology suitable for supporting multiple clients. With a growing number of clients, one may wish to deploy a network of multiple brokers (For e.g., 10000 / 400 = 25 brokers in the above scenario) so that all clients may receive acceptable audio / video transmission. Further, for fault-tolerance purposes, one may also want to have multiple links between brokers such that the failure of a subset of links may not crash the entire system. Finally, setting up of links becomes complicated if one or more brokers are behind restricted networks or in different administrative domains.

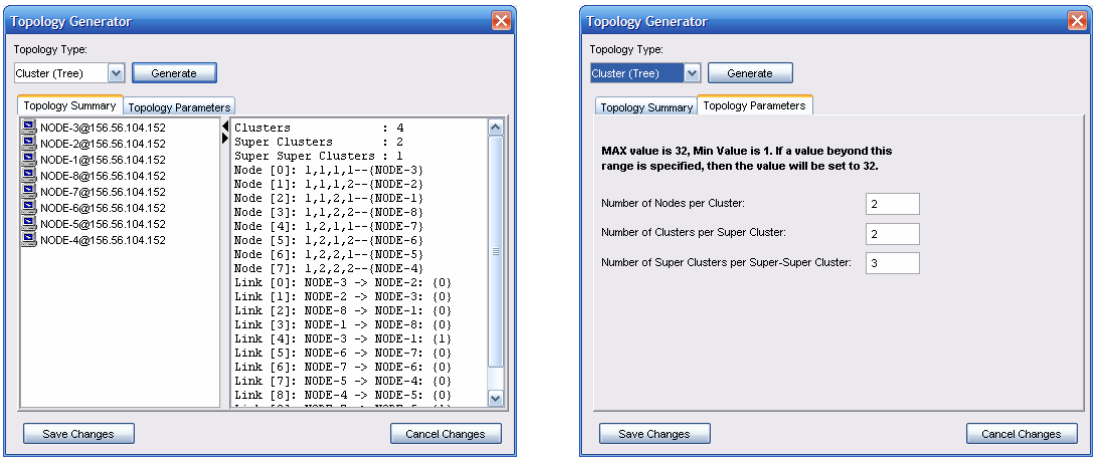
6.2. Management of Brokers

NaradaBrokering consists of multiple peers called brokers. These brokers can be connected in specific topologies, each topology providing varying degree of fault-tolerance and number of links. Further, connection between brokers can use different types of transports such as TCP, UDP and SSL among others. Selection of one of such available protocols (*"will tolerate missing packets but require scalability as in UDP"* vs. *"require a dedicated loss-less TCP connection"*) is application dependent and must be configured dynamically.

Further, setting up of a link between 2 brokers may involve a 3rd service and / or additional configuration entries. An example is UDP based P2P connection traversing NATs) in which case, a relay service is used to record each endpoint's public address. Another example is setting up an HTTP(S) connection which may require traversing an authenticating firewall. This in turn requires a *username* and *password* to authenticate the outgoing connection.

Finally the management architecture must also allow an administrator to shutdown existing brokers and bring them up again with a completely different configuration. We term these actions as "*Managing the Grid Messaging Middleware*".

6.3. Generating Broker Topologies



(a) (b)

Figure 20 Topology Generator GUI (a) Topology Summary (b) Topology Parameters

Brokers are dependent resources. This dependency is introduced by the fact that to establish a connection between brokers, both brokers must be appropriately configured and be up and running. If the target broker is down, then the source broker needs additional time to establish a connection. This delay adds to the overall deployment time of the topology.

The *Management GUI* provides an extensible *Topology Generator*. Currently we provide support for generating a *CLUSTER* topology and a *RING* topology. Figure 20 shows the GUI for the topology generator. The parameters for the selected topology can be set in the *"Topology Parameters"* tab. These parameters are used by the generator code while generating the topology. We now describe the two topologies and analyze these topologies in terms of deployment time.

6.4. Cluster Topology

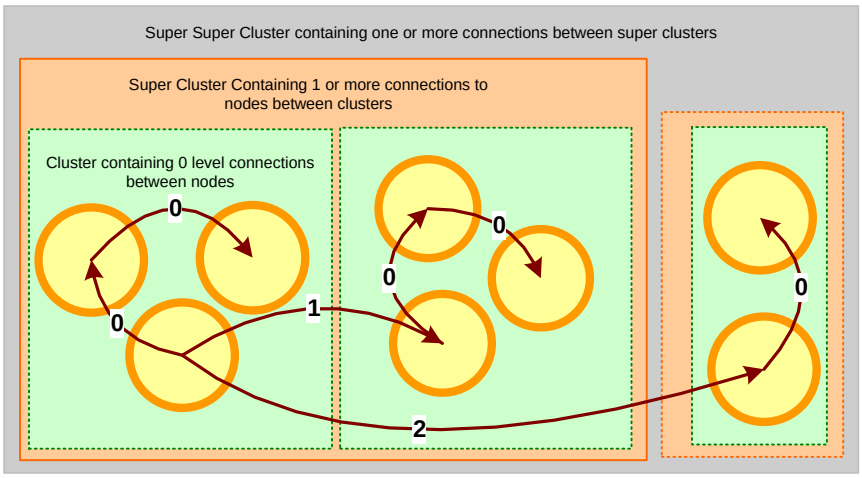


Figure 21 Cluster Topology

Figure 21 shows the first topology which is a cluster of brokers. This topology consists of a set of brokers connected in a *CHAIN* at the lowermost level (level 0 connections). The choice of CHAIN is arbitrary and any other topology may be used at the lowermost level. Two or more clusters may be combined at a super-cluster level. In the topology generator, the first node from each cluster is selected to be connected in a *CHAIN* at the super-cluster level. Finally, two or more super-clusters are connected to form a super-super-cluster at level 2.

NaradaBrokering allows a maximum of 32 nodes per cluster, 32 clusters per super-cluster and 32 super-clusters per super-super-cluster. Each node in the broker network is assigned a node address (Refer Figure 22) that aids in event dissemination. The node address which is represented by a set of integers such as **`<8, 2, 1, 4>`** must refer to a unique node in the network. Each level in the set is represented by an integer and since integers are usually 32-bit wide, we get 32 unique positions for each level. This causes a maximum of 32 entities per level.

Thus, node addresses **`<..., 2, 5>`** and **`<..., 2, 1>`** are invalid since the level 0 nodes have node address bits as **`5:[0101]`** and **`1:[0001]`** which has the last bit common and such node addresses are considered invalid during processing an address request.

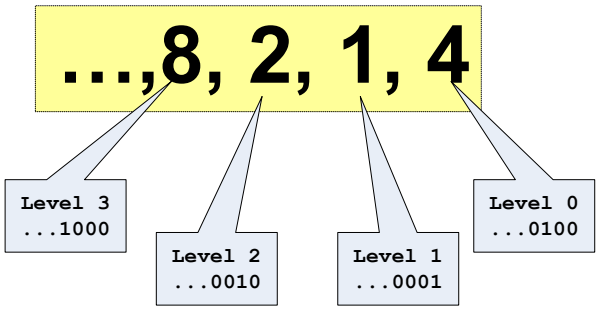


Figure 22 Anatomy of a Node Address

A sample node address assignment on the Grid Farm machines is shown in Figure 23. Here note that all nodes in the same cluster differ only at level 0. Similarly any 2 nodes in the same super-cluster differ only in the number at level 1.

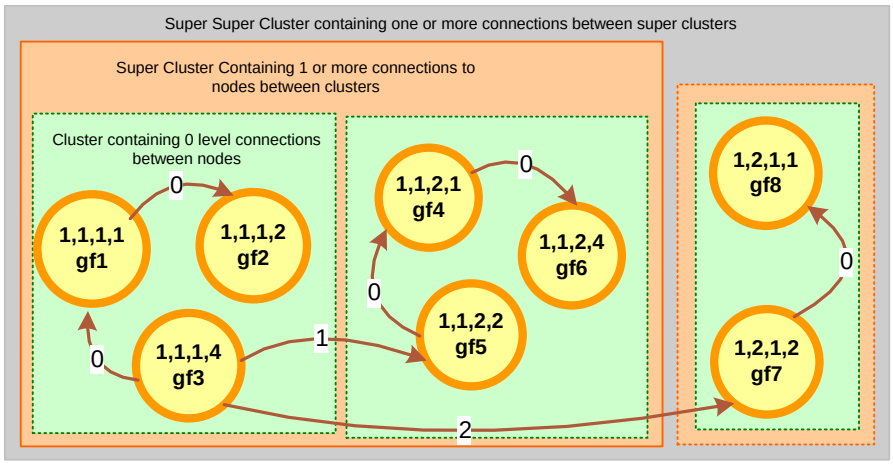


Figure 23 Sample assignment of Node address and cluster formation on Grid Farm machines

6.5. Ring Topology

The basic ring topology is illustrated in Figure 24(a). This topology is a level 0 interconnect of a circular linked list. With this topology we can have a maximum of 32 nodes per ring (since NaradaBrokering allows a maximum of 32 nodes per level). If there are more nodes in the ring, they will be connected at level 1.

Figure 24 (b) shows a sample arrangement with the actual node addresses. Both, the ring and cluster topology have an important characteristic that all nodes are present on un-firewalled and non-NAT'ed hosts.

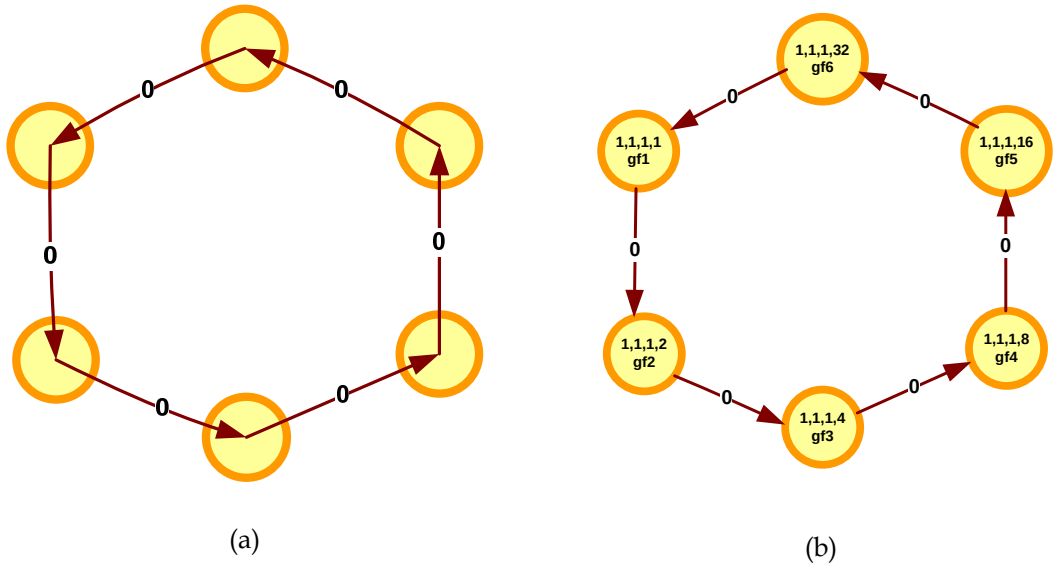


Figure 24 Ring Topology (a) Level - 0 Links (b) Sample node address assignment

6.6. NAT Traversal for Broker Connections

An important requirement of peer-2-peer communication is establishment of links directly between the peers in question. A general strategy (when 2 peers behind NATs wish to communicate) is to use a third peer in the outside (non – NAT'ed) network. This approach although being very simple to implement, suffers from scalability problem. As the number of distinct peers that wish to communicate increases, this third peer may easily become overloaded.

Another approach is to establish a direct link [87-89] between the peers. We use a relay server present on an *un-firewalled* node to aid the discovery of public/private IP addresses for creating links. A technique known as *hole-punching* is utilized to establish the link. This is illustrated in Figure 25. The connections between peers can then be setup using the steps illustrated in [90].

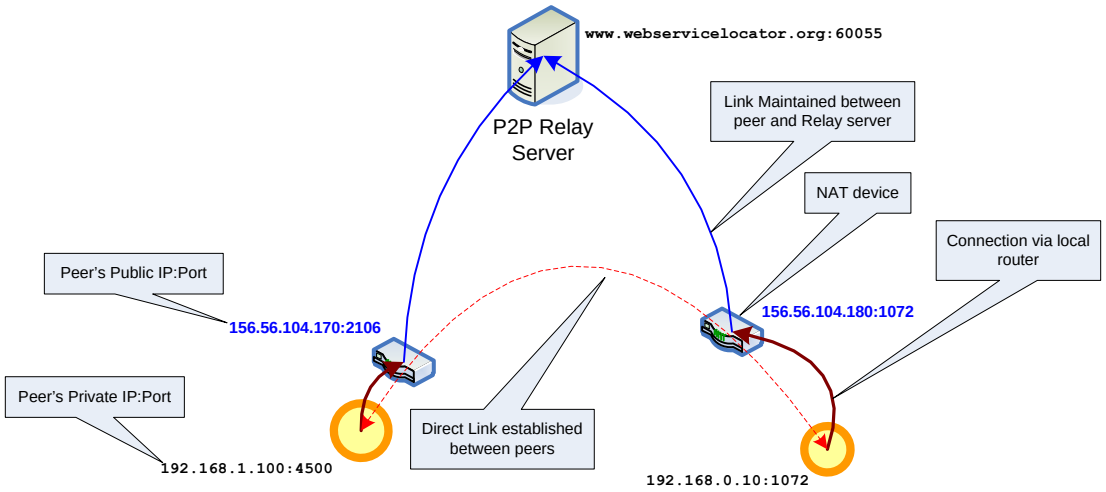


Figure 25 NAT Traversal for creating connections

We use a randomly generated UUID string to correlate multiple P2P connections. Once a connection is established, communication between the peers may be carried in normal fashion. The prototype contains an implementation of UDP hole punching. The connections can be established only when the NAT is a *cone* NAT rather than a *symmetric* NAT. In cone NATs the IP:PORT translation is preserved when originating IP:PORT is same. In symmetric NATs, a new public port number (sequentially incremented) is assigned for every new outgoing connection. In this case, port prediction techniques must be applied. Such techniques are being investigated in STUNT [91].

6.7. Policies

The management architecture provides 2 policies that determine the action to take on detection of failure. In general, *failure detection* is a non-deterministic process, i.e., it is impossible to distinguish deterministically between a *SLOW* process and a *FAILED* process. The management architecture, however, uses heartbeat and multiple retries to make a guess about the resource's status. If a resource cannot be contacted by any known means, it is considered as failed.

Policies are specified as a WS-Policy [92] document. The prototype currently defines the following policies:

6.7.1. Wait for user Input

The policy states that no action is to be taken. Rather, the resource manager will wait until the resource is re-instantiated via some out-of-band means. The resource manager will note that the resource in question is **UNREACHABLE** and update this status in the registry. Periodically, previously **UNREACHABLE** resources are reassigned to new resource manager processes and the new resource manager tries to establish contact with this resource. If successful, the management process continues as usual. If unsuccessful, the resource manager will rewrite the status as **UNREACHABLE** and exit. This policy is specified as follows:

```
<pol:Policy xmlns:pol=http://schemas.xmlsoap.org/ws/2004/09/policy
      xmlns:pol1="http://www.hpsearch.org/schemas/2006/07/policy">
  <pol:All>
    <pol1:RequireUserInput/>
  </pol:All>
</pol:Policy>
```

The system administrator / user may then suggest a different course of action to overcome the failure.

6.7.2. Automatically Instantiate

In this policy, the user specifies that the resource instance be automatically created. Towards this end, the user is responsible for specifying the location of the fork process daemon. The resource manager will try to re-instantiate the resource by sending an appropriate message to the fork process daemon. The policy may be specified as follows:

```
<pol:Policy xmlns:pol=http://schemas.xmlsoap.org/ws/2004/09/policy
      xmlns:pol1="http://www.hpsearch.org/schemas/2006/07/policy">
```

```
  <pol:All>
    <pol1:AUTOInstantiate
            forkProcessLocator="udp://156.56.104.152:65535"/>
  </pol:All>
</pol:Policy>
```

The failed resource is restarted with the basic default configuration such as the UUID and resource type. Once instantiated, the resource sends a message to the resource manager who then continues the management as before by configuring the resource to bring it up to the last known user-defined configuration.

A point to note about this particular policy is that, the fork process may not always be accessible by direct UDP connection, such as when the resource (broker) is behind a firewall. In such a case, the fork process may be contacted by having either the fork process connect to the system's messaging node and subscribe to a topic or routing the request to the appropriate fork process via a publicly visible service endpoint which has been appropriately configured.

Finally, it must be noted that when using this policy, consistency would be handled as per the discussion presented in Section 3.3.1.

6.8. Analysis of Broker Management

As mentioned earlier, the actual management work within the Manager process is done by a Resource-specific Resource Manager thread. The Resource Manager thread defines the interactions specific to the resource being managed and interacts accordingly with the Managee. System policies also play a major role on the degree of interaction between the Resource Manager and Managee. The number of interactions determines the scalability of management architecture. As the number of interactions increase, the time spent in performing an activity increases. This is typically the case with retrieving system state or committing state to the registry.

6.8.1. Interactions between Broker Manager and Broker Service Adapter

We define the interactions in 5 major groups, namely HeartBeat ($\mathbf{I}_{\mathbf{Heartbeat}}$), Create ($\mathbf{I}_{\mathbf{Create}}$), Delete ($\mathbf{I}_{\mathbf{Delete}}$), Modify ($\mathbf{I}_{\mathbf{Modify}}$) and Event ($\mathbf{I}_{\mathbf{Event}}$).

We outline the interactions for our prototype system which manages a distributed brokering network. Here the Resource Manager is the Broker Network Manager (BNM) and the Managee counterpart is the Broker Service Adapter (BSA). All interactions use 2 messages, 1 for Request and 1 for the associated Response. The Link Loss event and heartbeat message is only a notification style message from the BSA to BNM and hence only 1 message is involved. A receipt of this event or lack of one (e.g. heartbeat) may trigger one or more interactions defined above depending on system policy.

Message Type	From	To	Operation	When is it executed?	Time Interval (msec)	Num Msgs
$\mathbf{M}_{\mathbf{Heartbeat}}$	**BSA**	**BNM**	Heartbeat	Sends aliveness heartbeat	E.g. every 5 sec	1
$\mathbf{M}_{\mathbf{Retrieve}}$	**BNM**	**BSA**	Get Broker Information	Retrieves Broker State.	Policy Dependent E.g. every 10 sec	2
$\mathbf{M}_{\mathbf{SetConfig}}$	**BNM**	**BSA**	Set Configuration	Once at start and later ONLY when configuration change is requested.	- NA -	2
$\mathbf{M}_{\mathbf{CreateBroker}}$	**BNM**	**BSA**	Create Broker	Once at start and later ONLY when failure occurs or configuration change is requested.	- NA -	2
$\mathbf{M}_{\mathbf{CreateLink}}$	**BNM**	**BSA**	Create Link	Once at start and later ONLY when configuration change is requested. If there are K links, this is executed K times	- NA -	2
$\mathbf{M}_{\mathbf{DeleteBroker}}$	**BNM**	**BSA**	Delete Broker	Only when configuration change requested	- NA -	2
$\mathbf{M}_{\mathbf{DeleteLink}}$	**BNM**	**BSA**	Delete Link	Only when configuration change requested If K links need to be deleted, this is executed	- NA -	2

				K times		
$M_{LinkLossEvent}$	**BSA**	**BNM**	Event – Link Loss	When failure occurs	- NA -	1

Table 5 Interactions between Broker Service Adapter and Broker Network Manager

To approximately find the number of messages exchanged per second between each pair of BSA and BNM, we also define the following:

Symbol	Description
$F_{Heartbeat}$	Heartbeat frequency (usually 1/5) (once every 5 sec)
$F_{Retrieve}$	Frequency of retrieving broker state. Usually 1/10 (once every 10 sec)
K	Number of links originating from a given managed broker (initially created)
Z	Number of links that are created after modification

Then the various interactions result in the following messages:-

$I_{Heartbeat}$

$= M_{Heartbeat} * F_{Heartbeat} + M_{Retrieve} * F_{Retrieve}$

$= 1 * (1/5) + 2 * (1/10)$

$= 2/5$ messages/sec

$I_{Create} = (M_{SetConfig} + M_{Create} + K* M_{CreateLink}) = (4 +2K)$ messages

$I_{Modify} = [(K * M_{DeleteLink} + M_{DeleteBroker}) + (M_{SetConfig} + M_{CreateBroker} + Z * M_{CreateLink})]$

$= [(2K + 2) + (4 + 2Z)]$

$= 6 + 2K + 2Z$ messages ... (Deterministic state change)

$I_{Modify} = [M_{DeleteBroker} + (M_{SetConfig} + M_{CreateBroker} + Z * M_{CreateLink})]$

$= [2 + (4 + 2Z)]$

$= 6 + 2Z$ messages ... (OK, since deleting broker deletes links)

$I_{Delete} = (K * M_{DeleteLink} + M_{DeleteBroker})$

$= (2 + 2K)$ messages ... (Deterministic state change)

$I_{Delete} = M_{DeleteBroker} = 2$ messages ... (OK, since deleting broker deletes links)

$I_{Event} = M_{LinkLossEvent}$ messages

Here, When the broker is deleted, all the links are also deleted, so the term $K * M_{DeleteLink}$ is unnecessary, but may be included for cases where a deterministic state change is desired rather than an abrupt change. In the analysis that follows, we assume that cleaning up involves only deleting the broker i.e. 1 interaction (2 messages). Further, modification involves deleting the broker, setting configuration, creating the broker and creating outgoing links from the broker.

The above equations define the number of messages that occur between 1 BSA – BNM pair. If there are N managees (BSA) then,

Message rate to sustain for heartbeat messages

$= (I_{Heartbeat})*N = 2N/5$ messages/sec

For $N = 800$, the message rate is 320 messages / sec $<< 5000$, well within the max messages supported by a single broker. On the other hand, the broker will be overloaded with heartbeat messages when

$N \rightarrow 5000 * 5/2$

i.e. $N \rightarrow 12500$

An initial one-time cost exists for $N * I_{Create} = N*(4 + 2K)$ messages that are exchanged during the initial run for I_{Create}. For shutting down the entire system, an exchange of $N * I_{Delete} = N*2$ messages is required. Again for $N = 800$ and $K = 10$, the values are 19200 and 1600 respectively. With a broker supporting up to 5000 messages per second, we can conclude that the maximum overhead spent in message transit is up to 4 sec. The total time in creating / deleting $N (= 800)$ resources will be $4 + T_{Exec}$ seconds, where T_{Exec} is the time spent in retrieving data from registry and determining the configuration, setting configuration and for the broker to instantiate and create links. Typically $T_{Exec} » 4$ sec and hence we posit that message transit time is negligible in our system as compared to other costs and can be disregarded.

Let X be the number of Managees that report failure ($X <= N$) which requires the associated Resource Managers to take some or other action. Usually the action in case of broker network

would be to simply recreate the missing link, and the messages exchanged would be $X*2*Z$. Assuming that the action is to completely flush out the broker and recreate the broker and links (worst case), the number of messages exchanged is

$= X * I_{Modify}$

$= X * (6 + 2Z)$

Let T^X_Z be the time to transmit Z messages. If $T_{Recovery}$ is the average time spent in processing a failure message and determining the appropriate action, then the total time is

$Time = T^X_X*(3+K)$ request messages + $X * T_{Recovery} + T^X_X*(3+K)$ response messages.

$= X * T^X_{(6+2K)} + X * T_{Recovery}$

Again, $T_{Recovery}$ involves one or more interactions with the registry. We foresee that as the major cost involved and hence we ignore the $T^X_{(6+2K)}$ term.

6.8.2. When can T^X_Z be ignored?

For most of the analysis presented here, we chose to ignore the latency of data transmission .This is mainly motivated by the fact that $T_{Recovery} \gg T^X$. To see why, we define the following:

$T_{Process}$ - the time to process each request (Involves 1 or more interactions with Registry)

T_{Exec} - the time spent in actually executing the requested action. As a rule of thumb, 2 message exchanges are required for every action. Thus, when the total actions taken are $1+K$, total message exchanges is $2+2K$, and the total time spent is $(1+K)*T_{Exec}$.

$T_{A<->B}$ - the Average latency of data transfer between entities A and B.

Total time per broker

$= T_{Modify}$

$= T_{BSA<->BNM} + T_{BNM<->Registry} + T_{ProcessAtRegistry} + T_{Process(BNM<->BSA)}$

$= T^X_{(6+2K)} + T^X_{(2+2K)} + (1+K)*T_{ExecRegistry} + (1+K)*T_{Exec(BNM<->BSA)}$

Total messages exchanged $= (6+2K) + (2+2K) = 8+4K$

Here the term **(2+2K)** assumes that when there is **1** interaction, **1** request-response is required to do a status check with the registry. Thus, for **1** broker and **K** links, we require **(2+2K)** interactions.

Hence, the average messages exchanged per second is

$$= \frac{(8+4K) * 1000}{T^{x}_{(6+2K)} + T^{x}_{(2+2K)} + (1+K) * T_{ExecRegistry} + (1+K) * T_{Exec(BNM<->BSA)}}$$

For the sake of illustration, assuming the following typical values

$$T^{x} = 5 \text{ msec}, T_{ExecRegistry} = 5 \text{ msec}, T_{Exec(BNM<->BSA)} = 50 \text{ msec}$$

We get,

$$Rate = \frac{(8+4K) * 1000}{(6+2K)*5 + (2+2K)*5 + (1+K)*5 + (1+K)*50}$$

$$= \frac{(8+4K) * 1000}{30+10K + 10+10K + 5+K + 50+50K}$$

$$= \frac{(8+4K) * 1000}{95+71K}$$

K is the number of outgoing links from a broker and depends on topology. Typical values of **K** are from 0 to 3. Thus, the message rate for K = 0 is ~ 85 messages / sec and for K = 3 is ~ 65 messages/sec.

In each case, the broker can easily support the required message rate and hence we can conclude that the data transfer time is negligible and hence can be ignored. Alternatively, If there are **X** resources which report failure, then assuming that a single broker can transmit up to **5000** messages per second, the number of failures that can be sustained per second is **5000/85 ≈ 58** (in case of K = 0) or (**5000/65 ≈ 77** , in case of K = 3). T^X becomes significant when number of brokers failing per second is greater than this value. Further note that the above calculation is an approximation and does not take into account practical limitations of systems such as queuing of messages at the registry which would make $T_{ExecRegistry}$ **> 5 msec.** Brokers are dependent resources, i.e., when a broker A has an outgoing link to broker B, then for that link to be created, broker B must have been initialized and be ready to accept incoming connections. Failure of this condition would easily increase $T_{Exec(BNM<->BSA)}$ from **50 msec** to a few seconds.

6.8.3. Interactions with Registry

The system state is periodically offloaded to the registry. Interactions with registry also occur when a user requests a state change and hence the resource managers periodically poll the registry. In our system, query for system state happens via separate messages for each type of information element sought. Amount of data offloaded to registry varies depending on the resource involved. In our current implementation, we read / write to registry using separate messages. The main information that gets stored in the registry is summarized in the table below

Type	Information	Typical Size
Node Info	Information regarding a broker node	1 K – 2 K
Link Info	Information regarding links between brokers	400 - 500 Bytes
Resource Log	List of LogEntries for each type of resource. Thus, for each broker, there is a separate ResourceLog for the broker and a separate ResourceLog for each outgoing link from the broker	Varies depending on number of entries in the Log and size of each LogEntry.

Table 6 Resource-specific information stored in Registry

Thus, if a particular Resource Manager requires retrieving 10 different resource-specific configuration policies, it makes 10 different calls to the registry. Requests may be multiplexed in one request but has not been implemented for simplicity purposes.

The main interactions are as follows:-

1. From the point of view of Broker management, the resource manager retrieves the Resource-specific Broker Node information followed by **K** more calls to retrieve **K** link information objects. Thus, the total messages are $\mathbf{M_{ReadState} = 2*(1 + K)}$. The factor of 2 accounts for the asynchronous request – response message exchange.

2. A manager typically offloads his state in the registry (only the Resource Log) which accounts for $\mathbf{M_{WriteState} = 2*(1 + K)}$ messages. The current implementation follows the pattern shown in Figure 26.

3. A separate heartbeat thread keeps registering in registry the aliveness of the manager process. This involves $\mathbf{M_{RegisterRenew} = 2}$ messages. Thus, even when the Manager process is multithreaded, we still have a single heartbeat thread that renews its aliveness with the registry. The system assumes that if the manager process is alive and running, it is managing the assigned Managees. If at some point, the Managee is deemed **UNREACHABLE** (possibly after multiple retries and failure to establish successful contact), the associated Resource Manager thread updates the Managee's state in the registry. This accounts for **2K** messages (A Request-Response message for each of the **K UNREACHABLE** Managees).

4. The Manager process also periodically checks for presence of unmanaged managees whose management can be taken over if possible. Since this is either a one time cost or occurs only when an assigned Managee is **UNREACHABLE**, we may consider the cost negligible unless the MTBF (**M**ean **T**ime **B**etween **F**ailure) of the system is low.

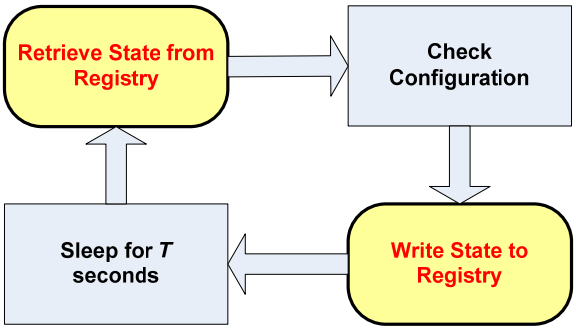


Figure 26 Registry Interaction

Thus, to summarize, the total interaction is given by

Total messages

$$= M_{ReadState} + M_{WriteState} + M_{RegisterRenew} + M_{GetManagee}$$
$$= 2*(1+K) + 2*(1+K) + 2 + 2D$$
$$= 4K + 2D + 6$$

Out of the above terms, (for a stable system), only the $M_{ReadState}$ and $M_{RegisterRenew}$ are executed at regular intervals. Current values are once every 10 seconds for reading state and once every 5 seconds for renewing via heartbeats. Hence, the message rate is

$$Rate = 2(1+K)/2 + 2/5$$
$$= 7/5 + K \text{ messages/sec}$$

Requests may be done using a separate TCP connection per request / response, sending messages over UDP or using the messaging node for sending requests / responses. In the current implementation, we use UDP transport for communication between Manager and Registry. However, this may very well be carried over a topic via the messaging nodes. An advantage of using a separate UDP transport mechanism is to reduce failure when messaging node goes down. Requests made over TCP require establishing a separate TCP connection every time a message is sent and is more costly as opposed to using UDP or messaging node for communication.

Apart from storing data, additional processing is required to adjust internal data structures. For example, when a manager process is assigned a resource to manage, the resource's service

adapter's UUID is stored in the Manager's internal tables for future access. Also, a search through all the records might be required to check if any Manager process has not renewed within the system specified time interval, in which case all the Managees managed by this Manager would be assigned to a separate manager process. As explained in Section 3.1.5, the registry is usually backed by some form of persistent storage to ensure fault-tolerance. Writes to registry are committed only when the persistent storage acknowledges the write. Reads can usually be served from an in-memory cache. Thus, writes are much costlier than reads. Hence, the goal is to minimize the writes to the registry. As an example, when the persistent storage is served by a WS-Context services, the *writes to registry* take about **`45 msec`**, while reads can be served within **`30 msec`** (without database access) - **`37 msec`** (with database access). Ref [93] discusses the interactions when using a WS – Context service implementation of the registry.

The definition of a resource state is highly dependent on the resource in question. From the point of view of broker management, only the user commits Node and Link Information. This information is static unless the user specifically requests a change in the configuration of managed brokers. Once submitted, it is assumed that this information remains static over a long period of time. The Managers only write to registry when the Resource Log has changed. For a relatively non-varying system, the Resource Log will not change for a long time. Thus, writes occur less frequently.

6.8.4. Managee – Registry Interaction

When Managees are instantiated, their associated Service Adapters register themselves in the registry. Unless failure occurs, the registration involves 2 messages (a registration request and associated response). Future heartbeat messages are sent to the Resource Manager and so also all events associated with the functioning of the Resources. If a Registration request message fails to reach the Registry, the Service Adapter tries after waiting for a small interval (usually 2 seconds)

and tries until a successful registration has occurred. Registration process is idempotent, so duplicate registration is acceptable.

Note that this is a required step as only the registered Service Adapters may be managed by the management framework. In our current implementation, the registration process simply involves, storing the Resource metadata. In future, the Service Adapter may optionally present security credentials such as digital certificate, username/password or some other form of valid credentials, so that only valid (authorized) resources would be registered.

6.9. Benchmarking Topology deployment

To benchmark the time it takes to deploy a topology from scratch, the topologies are created and committed to the registry. The manager process executes a registry check routine every 2 seconds. This interval is configurable and would affect the time it takes for the deployment process to start. Further, not all resources (broker service adapters) would have been initiated at the same time, so the time at which the registry is read for presence of updated resource configuration (Node / Link Information) is different. This implies that a node may be initiated before a node it is dependent on. This causes delays in deploying the overall broker network.

We utilize the concepts and equations defined in Chapter 5 when computing the deployment and recovery (after failure) times.

6.9.1. Resource State Size

The state size per resource is very dynamic and is very resource-specific. As listed in Table 6 the objects that make up the broker state comprise of the following:-

1. NodeInfo object – Stores Node information such as Broker Configuration and outgoing links (approximately 2 Kbytes);

2. LinkInfo object – Stores details regarding the link, such as protocol to use, destination host and port of the broker to which the broker in question will connect to (one per outgoing link), approximately 512 bytes;

3. Resource Log (log of entries, varies as per the number of logging entries) – With 128 Byte string entry and 16 entries, the size is about 2 Kbytes and

4. BrokerInformation – The current state of broker that is regularly loaded from the broker. This varies depending on number of outgoing links and is approximately 1 Kbyte with 5 links.

So, assuming 5 links, the size of state per thread is

$$= 1 * NUM_{NodeInfo} + 5 * NUM_{LinkInfo} + (1+5) * NUM_{ResourceLog} + 1 * NUM_{BrokerInformation}$$

$$\approx 1*2048 + 5*512 + 6*2048 + 1*1024 \text{ bytes}$$

$$\approx 16 \text{ Kbytes}$$

Thus, the runtime state size of the Broker Resource is small and hence the Broker is an ideal example of a Resource that can be managed using our management architecture.

6.9.2. Initialization Costs

In this case, we note the amount of time it takes for a broker management system to perform actions such as starting /shutting the broker, setting configuration and creating / deleting links. Table 7 lists the time required for the operation to take place the first time. The time required after the system has been initialized is shown in Table 8.

	Mean	Standard Deviation	Maximum	Minimum	Standard Error
Set Configuration	778.0	15.47	812.0	761.0	5.0
Create Broker	610.1	19.3	643.0	591.0	6.1
Create Link	160.5	7.3	174.0	153.0	2.3
Delete Link	104.5	4.7	111.0	98.0	1.5

Delete Broker	142.0	4.2	149.0	136.0	1.3

Table 7 Recovery after Failure (non-initialized state)

	Mean	Standard Deviation	Maximum	Minimum	Standard Error
Set Configuration	33.90	10.17	52.00	22.00	3.22
Create Broker	56.67	6.18	67.00	48.00	2.06
Create Link	26.67	5.66	36.00	20.00	1.89
Delete Link	20.11	4.08	28.00	13.00	1.36
Delete Broker	129.20	6.14	141.00	118.00	1.94

Table 8 Time (initialized state) required per operation (msec)

The time required per operation in initialized state is significantly less as compared to in the non-initialized state because the JVM needs to load the appropriate classes the first time they are referenced and this takes more time. Once initialized, further modifications take lesser time.

Further, note that deleting a broker takes approximately the same time. This is because, in both cases, no new classes are loaded to delete a broker object and this merely represents a cleanup of existing objects.

6.9.3. Ring Topology

The Ring topology contains **N** nodes and exactly **N** links (one per each node). Thus, applying the formulae in preceding sections, we get

State per management Thread:

$$= 1 * NUM_{NodeInfo} + 1 * NUM_{LinkInfo} + (1+1) * NUM_{ResourceLog} + 1 * NUM_{BrokerInformation}$$

$$= 1*2048 + 1*512 + 2*2048 + 1*1024 \text{ bytes}$$

$$= 7.5 \text{ Kbytes}$$

Each thread needs to load 2 objects from registry and write 2 objects to registry while committing state. Thus,

$T_{ReadState} \approx 2 * 5 = 10$ msec

$T_{WriteState} \approx 2 * 5 = 10$ msec

This assumes that each read / write operations requires **5 msec** to complete. On failure, the resource endpoint (the broker service adapter, in our case) needs to be spawned. This entails, sending an appropriate message to the configured Fork Process to spawn the process. Once a BSA is spawned, a node in a ring topology can be brought up within

$T_{Correction}$

$= T_{ReadState} + T_{Restart}$

$= T_{ReadState} + (T_{SetConfig} + T_{CreateBroker} + T_{CreateLink})$

$= 10 + (778 + 610.1 + 160.5)$

≈ 1548 msec

6.9.4. Cluster Topology

The cluster topology with **N** nodes is constructed as follows:

Each cluster is a chain of nodes. Thus, there are **(C – 1)** links for **C** nodes. For each node except the last, there is one outgoing link per node. Further, multiple clusters are connected in a chain to form a super-cluster. For this purpose, we select the first node from each cluster. Thus, the nodes connecting clusters have a maximum of **2** links each. Similarly, the nodes selected to connect super-super-clusters have **3** links each. Thus, the number of outgoing links varies from **0** to **3**. Table 9 shows the state that is maintained per management thread and the time required for reading /writing the state.

Number of Links	State (Kbytes)	$T_{ReadState}$	$T_{WriteState}$
0	1 * 2048 + 0 * 512 +(1+0)*2048 + 1 * 1024 = 5.0 Kbytes	1 * 5 = 5 msec	1 * 5 = 5 msec
1	1 * 2048 + 1 * 512	2 * 5 = 10 msec	2 * 5 = 10 msec

	+(1+1)*2048 + 1 * 1024 = 7.5 Kbytes		
2	1 * 2048 + 2 * 512 +(1+2)*2048 + 1 * 1024 = 10.0 Kbytes	3 * 5 = 15 msec	3 * 5 = 15 msec
3	1 * 2048 + 3 * 512 +(1+3)*2048 + 1 * 1024 = 12.5 Kbytes	4 * 5 = 20 msec	4 * 5 = 20 msec

Table 9 State and initialization time per management thread

Thus, on failure, a single node in a cluster topology can be brought up within

$$T_{Correction}$$

$$= T_{ReadState} + T_{Restart}$$

$$= T_{ReadState} + [T_{SetConfig} + T_{CreateBroker} + (T_{CreateFirstLink} + T_{AdditionalLinks})]$$

Thus, for nodes with no links, we get

$$T_{Correction}$$

$$= 5 + 778.0 + 610.1$$

$$\approx 1393 \text{ msec}$$

and for nodes with 3 links, we get

$$T_{Correction}$$

$$= 20 + 778.0 + 610.1 + 160.5 + 2* 26.67$$

$$\approx 1622 \text{ msec}$$

6.9.5. Results A: Recovery Costs for a single Resource (Broker)

In this test, we benchmark the actual time it takes to create a broker from scratch after it has failed. For the purpose of testing, a static broker is assumed to exist. The managed broker creates links to the static broker and we time recovery after failure. A step-wise procedure is outlined below:

1. The resource manager sends a shutdown message to broker which kills itself on receiving the message.
2. The resource manager times out, and starts a timer before initiating any action. Then it reads state of broker from registry.
3. A message is sent to the registered fork-process daemon to restart the failed resource. This step is required if the death of a broker also implies that its resource adapter is dead, in which case, the recovery process involves additional cost of recreating an instance of the broker service adapter.
4. The broker service adapter starts and registers itself in the registry and with the broker manager.
5. The broker manager then sets configuration, instantiates the broker object and creates a link to a known static broker.
6. After the link is created, a full recovery of broker has occurred, and we stop the timer started in step 2 above.

The results are presented in Table 10.

	Average (msec)	**Standard Deviation (msec)**
Spawn Process	**2362**	**56.71**
Read State	**7.7**	**2.54**
Restore (1 Broker + 1 Link)	**1420.5**	**27**
Restore (1 Broker + 3 Links)	**1615.8**	**258.72**

Table 10 Observed recovery time for a single broker

Thus, we note that the time to reload state from registry is about 8 msec while the time to do a recovery is about 1420 msec. If the resource recovery requires creation of a service adapter as a

front end to creating a resource an additional time is required to spawn the process and receive confirmation. This increases the cost by about 2400 msec.

6.9.6. Results B: Topology recreation costs for a set of Resources (Topology of Brokers)

In the second test, we construct topologies of brokers and time the topology re-creation time. Thus, in this case, we manually shutdown brokers and recreate the topology with the specified criteria. Once the resource-specific information is read, the broker network manager uses this information to configure the broker as per the user defined requirement. The testing methodology used is as follows:

A test accumulator sends a "*SHUTDOWN*" message to all brokers and starts a timer. After the first broker manager reports that it is starting correcting things another timer is started. When the last broker is successfully instantiated, the time difference is calculated. Thus,

Overall: Time from sending *SHUTDOWN* message to getting the last broker instantiated.

Total: Time from first broker instantiation start to last broker instantiation done. Instantiation is defined as achieving the exact configuration as set by user i.e. broker + required links

Then, Difference = Overall - Total = Response time for manager to detect failure and start taking action.

Number of Nodes	Total Links in the network	Overall Time (msec)	Total Time (msec)	Difference (msec)
Ring:				
8	8	12515	12372	143
4	4	8906	8224	682
1	(NO LINKS)	1156	770	386

Cluster:				
8	7	15968	15937	31

Table 11 Observed Times for deploying network of brokers

Table 11 lists typical system startup times. For the purpose of testing, the broker network manager checks the health of broker and its configuration every 2 seconds. This time is highly dynamic and is dependent on when individual components get initialized. Note that the observed time is highly dependent on the state of the broker topology when the links are created. Thus, if a broker A connects to broker B and broker B was not initialized and ready to accept connections, then broker A waits for a certain period of time before retrying connection. This would increase the time to deploy a topology as the number of brokers and hence the inter-dependency increases.

6.10. Discussion

The tests indicate that the failure detection can usually be done pretty quickly and failed state is restored within a reasonable time frame. Note that, as the number of brokers requiring management grows, the 2 second interval for successive health checks becomes impractical if each health check interval involves one or more reads from the Registry.

The Registry can only service a finite amount of request per second [93]. As an example, assuming 200 requests per second can be serviced by the registry, and if a particular broker manager requires 5 requests, then the system limits the maximum managers being served to 40. This factor limits the scalability of the system. Further, if the Registry is busy servicing initial requests, future requests can easily get queued and their associated responses may get delayed resulting into timeouts and thrashing, as resource managers spend more time getting responses from registry rather than getting any useful work done.

One way to get over this hurdle is to make registry requests at a lesser frequency. However, we note that if the operation is executed less frequently (e.g. every 2 minutes) then detection of failure would be slower. The detection of failure is fast when the process executes the *"Read State"* operation frequently. The choice is dependent on the desired quality of service and resource requirements and thus this parameter is resource dependent.

Chapter 7. Conclusions and Future Work

In this dissertation, we have presented a scalable, fault tolerant management framework. To make the management framework extensible we employed a service-oriented architecture based on WS-Management. WS-Management was selected because of its simplicity and also because we could leverage the WS-Eventing support recently added to the NaradaBrokering core.

A proof-of-concept implementation on how management can be utilized in managing NaradaBrokering was demonstrated and the prototype was evaluated. By enabling a GUI based management tool based on the management framework, broker topologies can be dynamically deployed and modified as required by a particular application. We believe that appending management capabilities to existing NaradaBrokering framework would promote interesting ways of utilizing NaradaBrokering's capabilities. We feel this is an important contribution to the NaradaBrokering system as the management framework not only provides ease of deployment of

brokers but also maintains the runtime configuration in a fault-tolerant manner transparent to the administrator of the deployed system. Such a capability is important for providing continuous access to clients when failures occur because of causes beyond human control.

The management framework provides maximum scalability when the runtime state required for the resource being managed is small. We define small as being one which can be written to or read from a registry using minimum number of interactions. When this condition is satisfied, the use of proposed management architecture is feasible. Our experimental evaluation shows that the management architecture adds about 1% additional processes to provide fault-tolerant management for a large number of resources. This feasibility is a result of use of NaradaBrokering itself as a scalable messaging substrate for communication. Further by using NaradaBrokering, resources behind firewalls can be managed by having them simply connect to the domain specific messaging node using tunneling protocols supported in NaradaBrokering.

7.1. Summary of Answers for Research Questions

We summarize the answers of the research questions presented in Section 1.3.

7.1.1. How can we build a fault-tolerant management architecture?

The system has been designed to be fault-tolerant in face of failure. The overall fault tolerance is the result of fault tolerance of individual components. Thus, the management framework is fault-tolerant by definition by the use of a hierarchical bootstrapping process which ensures that the system is continuously up and running when components fail. Management is fault-tolerant as the system state maintained within each resource manager is small as a result of which, on failure, a new resource manager may be separately instantiated and management continued from the point of failure. Finally, the resource management is itself fault-tolerant by use of such policies wherever applicable. As an illustration, we showed in Section 6.7.2 the

`AUTOInstantiate` policy used by the broker network that automatically instantiates a new resource instance when failure of managed resource is detected.

7.1.2. Can the management framework be made scalable?

Scalability of the system is by design. The overall management framework is arranged hierarchically (Refer Section 3.1.1) which allows scaling in a wide area deployment. For local scalability, we use a *"publish-subscribe"* based scalable messaging substrate. Our empirical results show that typically a single manager process can handle up to 200 requests with a linear increase in response time. Further a single manager process only needs to maintain a very few number of physical connections with the messaging node with respect to the number of resources it is managing. Communication is enabled by *publishing/subscribing* to appropriate topics. Finally our results show that with such a setup, a large number of distributed resources can be managed by an additional 1% extra resources. This makes the system scalable.

7.1.3. If such a system can be built, what is the overhead of such a system and is this overhead acceptable?

The management framework introduces several key components apart from the actual resources being managed. These key components (Resource Managers, Registry, Bootstrap Nodes, and Messaging Nodes) impart scalability and fault-tolerance to the management of the target resources. When the resource satisfies the criteria of a small runtime system state (Refer Section 5.9), we empirically proved that management can be done in a fault-tolerant fashion using about 1% more processes with respect to the number of resources being managed. This overhead is quite acceptable for managing large number of distributed resources.

7.1.4. How can we enable global management of resources (i.e. when access to resources may be restricted by presence of firewall and NAT devices)?

Remote management is enabled in the system by leveraging a "*publish-subscribe*" based messaging system which imparts *Location Transparency*. The resource to be managed is wrapped with a Service Adapter that is capable of communicating with a messaging node that is used to deliver management related messages from the resource manager to the resource in question. When resources are behind restricted networks such as due to NAT and firewalls, the Service Adapter is responsible for establishing a connection to the local messaging node. Use of NaradaBrokering based messaging node makes the delivery of messages between concerned entities impervious to network restrictions in most cases. The system leverages NaradaBrokering's firewall tunneling capabilities in presence of firewalls. NaradaBrokering supports a variety of transports such as TCP, UDP, HTTP, and SSL which may be used to connect to the local messaging node when other transports are restricted by firewalls. When resources are behind NATs, the Service Adapter will connect to the local domain's messaging node (which is typically in a non NAT'ed network with respect to the resource in question) and management can be enabled.

7.1.5. Can the management system be made interoperable and extensible?

The management system implements a service oriented management protocol, specifically WS-Management. Extensibility of management is due to the design of WS Management specification which does not restrict users from defining their own management actions apart from the default provided actions. Interoperability is achieved since any WS Management capable client may invoke appropriate management actions on the target resource.

7.2. Management of General Grid Services

Grid applications are composed of multiple services possibly running in different administrative domains. As defined in [94] the primary benefits are *"coordinated resource sharing and problem-solving in dynamic, multi-institutional Virtual Organizations"*. Here we discuss some of the key properties of our architecture and how they can be beneficial in the Grid computing community:-

1. The architecture is scalable for wide area deployments. This makes it suitable for managing services which are distributed over a wide area such as encompassing multiple administrative domains of different participating institutions (service providers).

2. By leveraging multiple transport feature of NaradaBrokering, we can manage resources which might be behind restricted networks. The management framework can be configured using domain specific policies. This would help managing resources that fall under disparate administrative domains without violating the resource provider's policies.

3. The architecture implements a service-oriented management approach. This makes the management extensible which simplifies addition of newer management capabilities. Use of Web-service based management protocols makes the management system interoperable. This would enable existing or future management architectures written using different languages or developed on different platforms to seamlessly manage these resources.

4. The scheme proposed in this thesis enables fault-tolerance of management architecture and of managed resources as illustrated in Chapter 6.

We believe that the above mentioned properties make the architecture suitable for enabling management of resources, possibly belonging to multiple institutions or different administrative domains and enforcing a variety of (access and security) policies.

7.3. Future work

Our current implementation and the feasibility of the system are based on the following conditions that the managed resource and its manager must satisfy:

1. The external runtime state required to be maintained per resource is small.
2. This state can be read / written from/to registry and resource in a small number of calls

These conditions are true for loosely coupled resources where the resource can be bootstrapped using some minimal configuration, and the resource is well-equipped to pursue further configuration / initialization.

In future, we would like to apply this framework to broader domains. We believe that this dissertation can spin off further research when the management framework is applied to in broader domains. This can bring up many interesting research issues, specifically challenging scalability of the system.

The system does not implement any security at this point. Thus, a malicious process may arbitrarily invoke operations on the resource which may result in the resource being moved to an unexpected state and not meeting its desired quality of service. As part of future work, we plan to look into implementing WS – Security [95] based security or NaradaBrokering's security infrastructure as outlined in Section 3.3.2.

Finally, more comprehensive metrics need to be taken into account when deploying manager processes. Current solution distributes manager processes in a round-robin fashion over the available set of participating nodes. Future solutions would also take into account system metrics such as CPU utilization, available memory and locality when deploying a new manager process or assigning resources to manager processes.

APPENDIX – A: THE MANAGEMENT GRAPHICAL USER INTERFACE

In this appendix we illustrate the Broker Management Graphical User Interface with screen shots and describe the working of the prototype.

1. Main Window

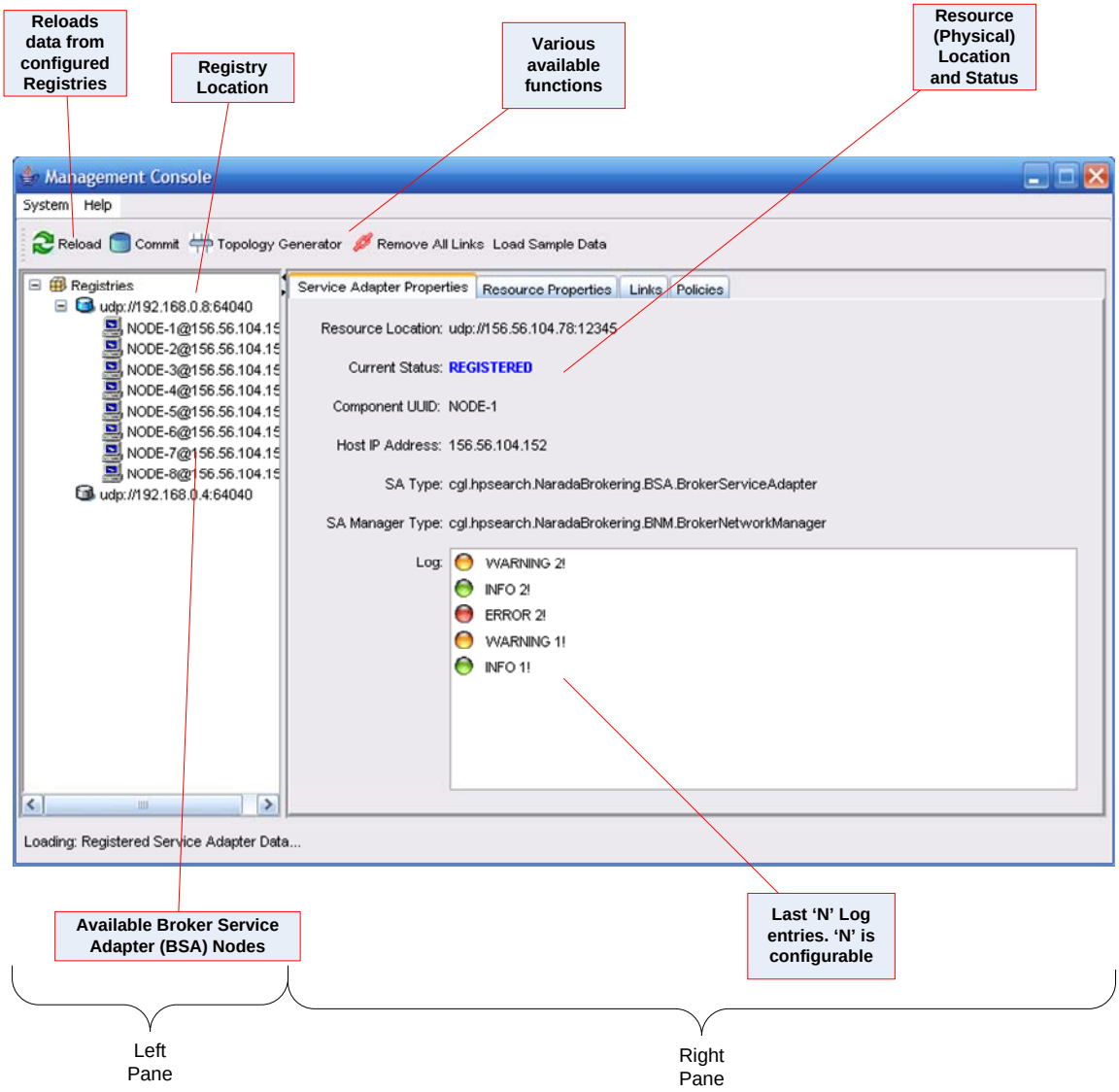


Figure A1 Main Window

The main window (Figure A1) shows a list of available Broker Service Adapter nodes and their associated registries. By selecting a node from the tree (left pane), one can view / set properties specific to the selected resource. The right pane consists of various resource-specific tabs for configuring the selected resource.

The *Reload* button on the toolbar, reloads data from the registry. This overwrites the current user state and configuration.

Commit button is used to save all changes to the registry.

The *Topology Generator* button starts the topology generator which uses default topology generation algorithms. If a user specific topology is desired, then the *Links* tab can be used to create and deploy a user-defined topology. Currently the Topology Generator provides support for 2 topologies *RING* and *CLUSTER*.

Remove All Links deletes all current links from the registry after the next commit.

The *Load Sample Data* is for debugging purposes to check the User interface functionality.

We now discuss the various tabs and functionalities of the GUI. The functionality depicted here is very specific to Broker Management.

2. Resource Properties

Figure A2 shows the main Resource properties window.

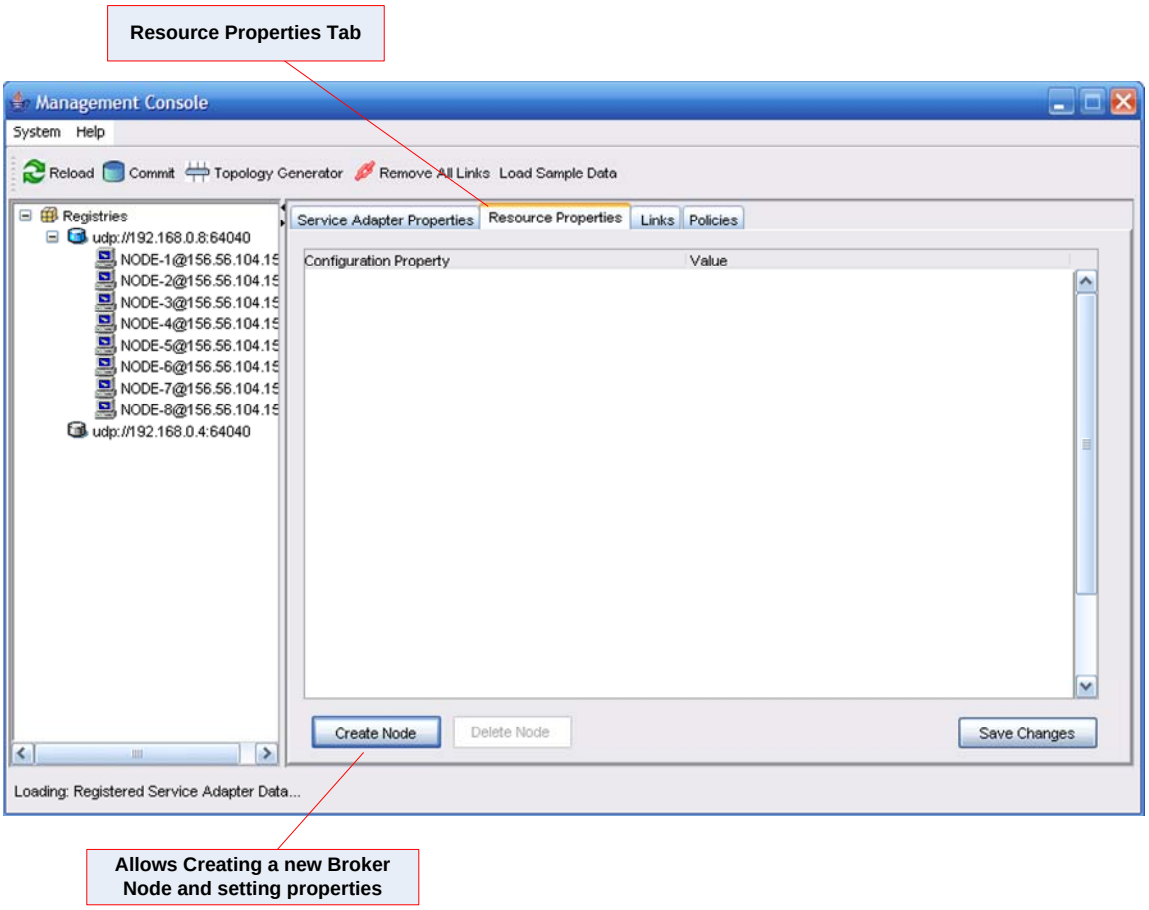


Figure A2 Resource Properties

The resource properties tab shows an editable table of *Configuration Properties* and their *Values*. Currently new values cannot be created. However, existing values can be edited. This allows a user to configure a broker node to run specific services (such as, Run TCP and UDP transport on specified ports but do not run HTTPS/SSL and HTTP etc…).

The first step is to create a new node and assign the default values. This can be done by clicking the "*Create Node*" button. Figure A3 shows the default configuration properties after creating a new node.

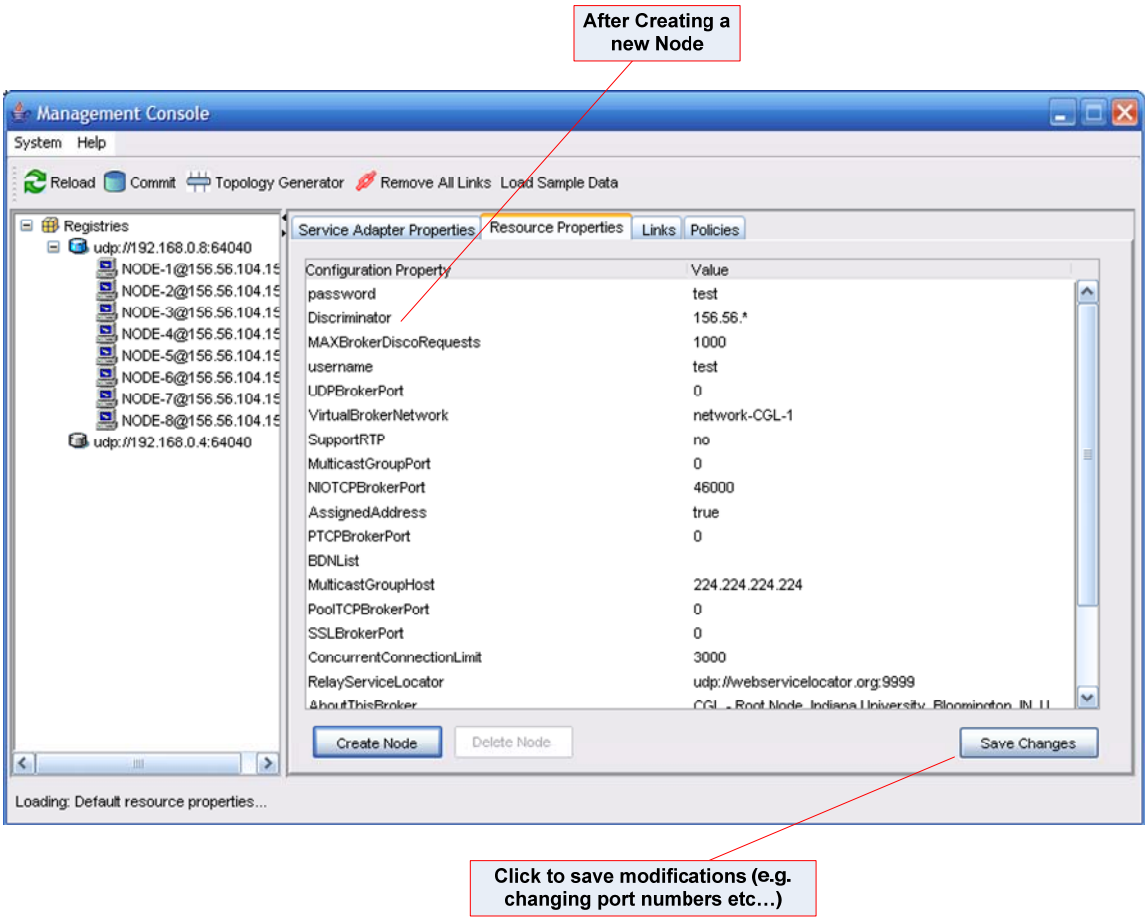


Figure A3 After creating a new node

To make any changes, simply double click the "*Value*" and press "*Enter*" when done. Finally, the changes to a node's configuration may be saved (on the user's side) by clicking "*Save Changes*".

3. Policies

Failure of nodes would cause the application using the broker to function erratically. Usual method is to re-instantiate a new broker process manually. Whenever possible, this may be automated by setting the appropriate policy. The default policy [Figure A4] is to wait for "*User Interaction*" which simply put, "*Does Nothing*".

An alternate policy [Figure A5] is to use one of the *Fork Process Daemons* to spawn a broker process and use the newly spawned process in lieu of the failed broker process. The following **MUST** be noted for using this feature.

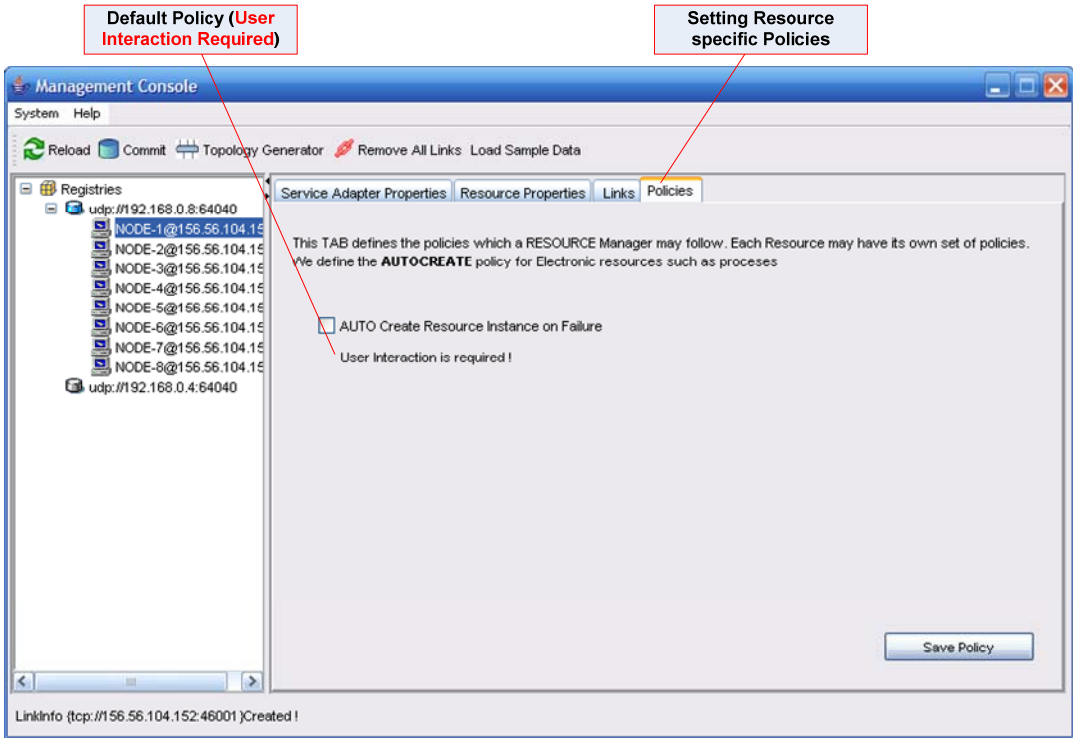


Figure A4 Default Policy (Require User Interaction)

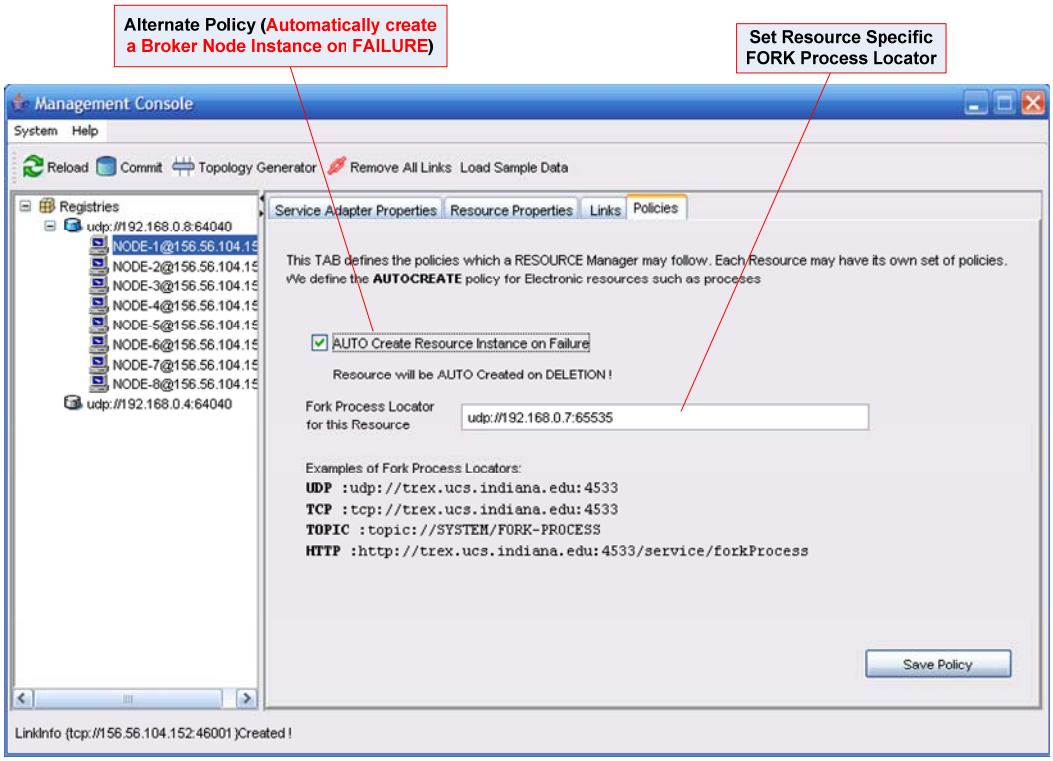


Figure A5 Alternate Policy (Automatically spawn a new Broker)

In the current prototype implementation, **Only** *Fork Process Daemons* directly accessible (via UDP / TCP /HTTP or via a NB topic) can be used to spawn a new process.

A failed process is typically indistinguishable from a *SLOW* process. Thus, the conclusion of a process failure is solely dependent on missing heartbeats and the inability of the manager to successfully establish a contact with the target resource after several retries.

4. Generating Topologies

The *Topology Generator* button on the toolbar starts the topology generator module. Currently we have implemented a *RING* and a *CLUSTER* topology generator. Each of these topologies have specific characteristics and may be used in specific situations. The main window for the topology generator is shown in Figure A6.

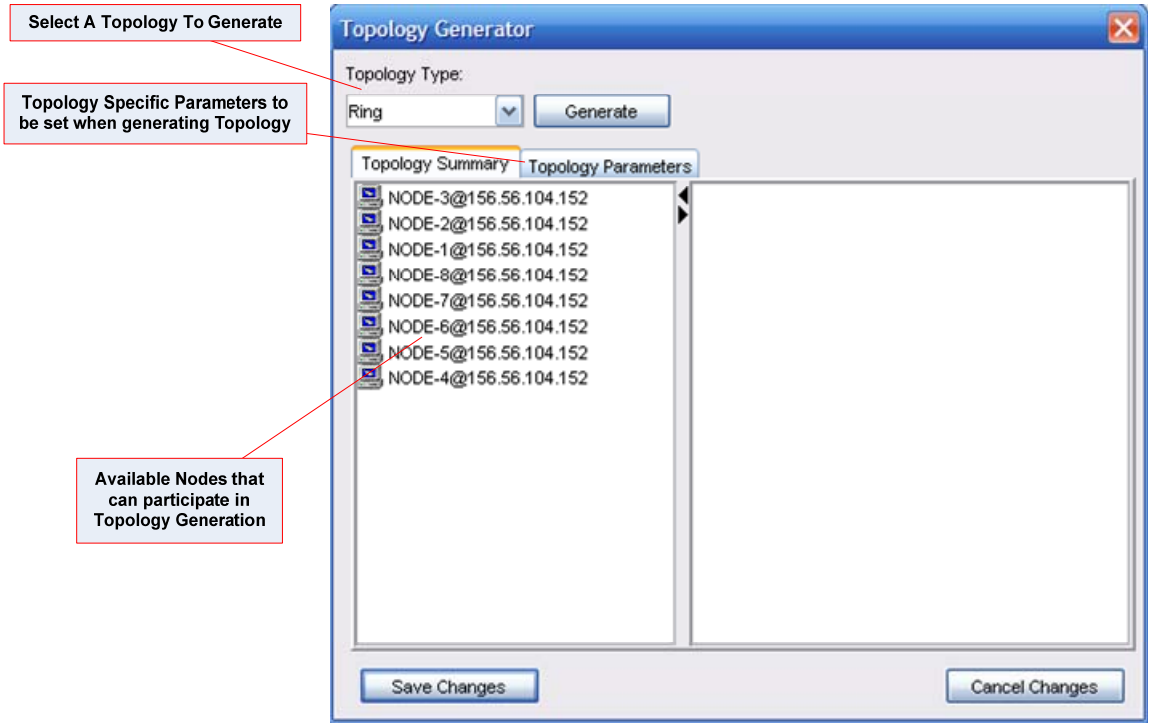


Figure A6 Topology Generator

On the left side is a list of available nodes. An *Available Node* is defined as a node which was *"created"* using the *"Create Node"* on the *Resource Properties* page. Such a node is assumed to be completely configured as any changes to this node after the links generation process would result in an incorrect deployment of the broker topology.

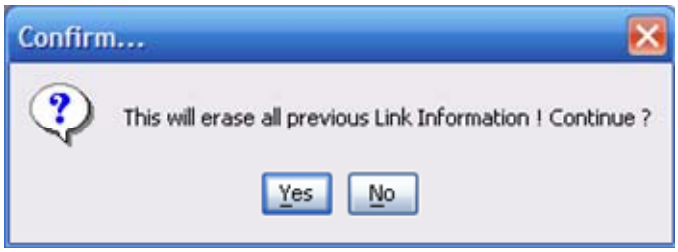


Figure A7 Warning Dialog asking for link deletion confirmation

The type of topology to generate can be selected from the drop-down list and after setting topology specific parameters on the *"Topology Parameters"* tab, the user clicks *"Generate"* to

generate the topology. The topology generation deletes all previous links and creates new links. A warning is issued (as shown in Figure A7) before the topology generation is started.

We now show the *RING* and *CLUSTER* topology generation on a sample data set.

Ring Topology

The Ring topology does not have any major topology specific parameters. When deploying broker network involving brokers behind NAT devices, a third party relay server (present in a non-NATed network) is used. The location of this server may be configured here as shown in Figure A8.

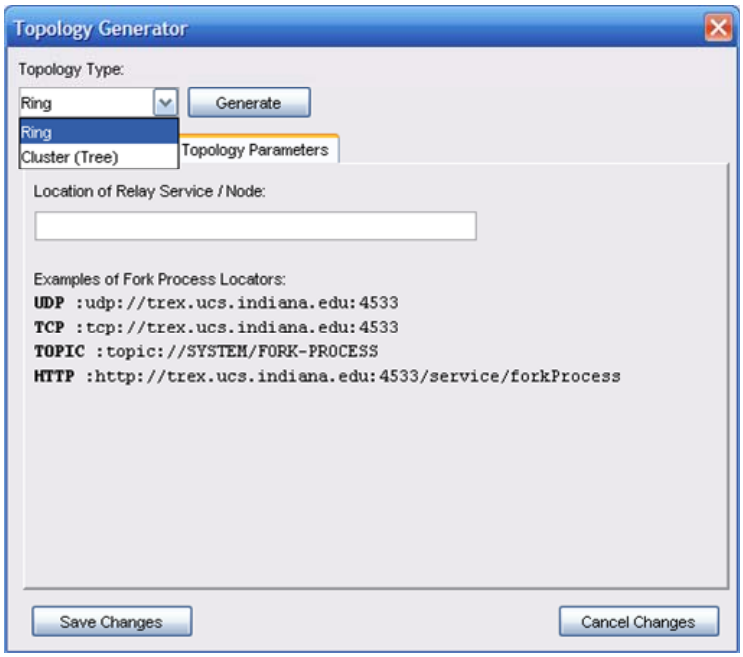


Figure A8 Ring topology parameters

The generated topology for an 8-node network is shown in Figure A9.

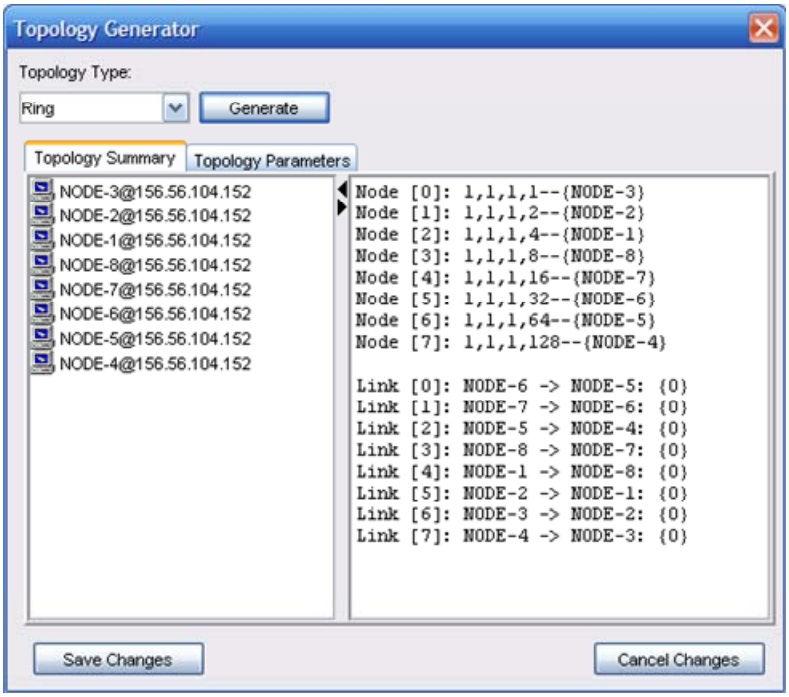


Figure A9 Nodes and Links Configuration for RING topology

Cluster topology

Cluster topology has more configuration parameters than the basic RING topology. These parameters [Refer Figure A10] define the characteristics of the generated topology such as number of clusters, super-clusters and super-super-clusters.

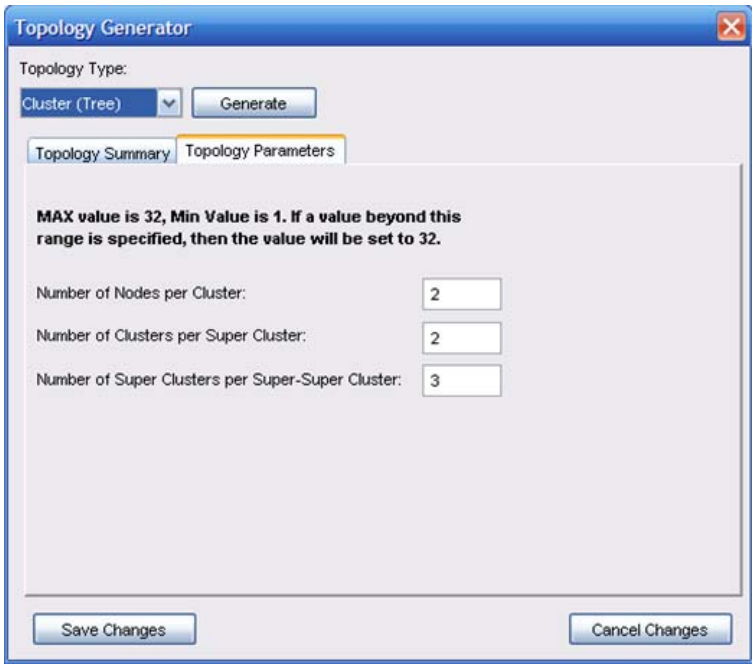


Figure A10 Cluster topology parameters

A sample cluster generated for an 8 node network using the above set parameters is shown in Figure A11.

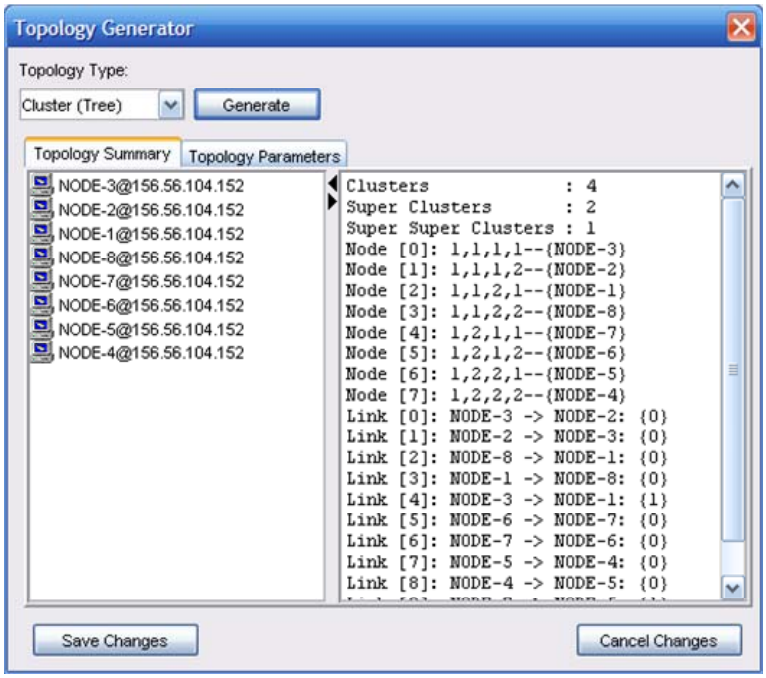


Figure A11 Nodes and Links Configuration for CLUSTER topology

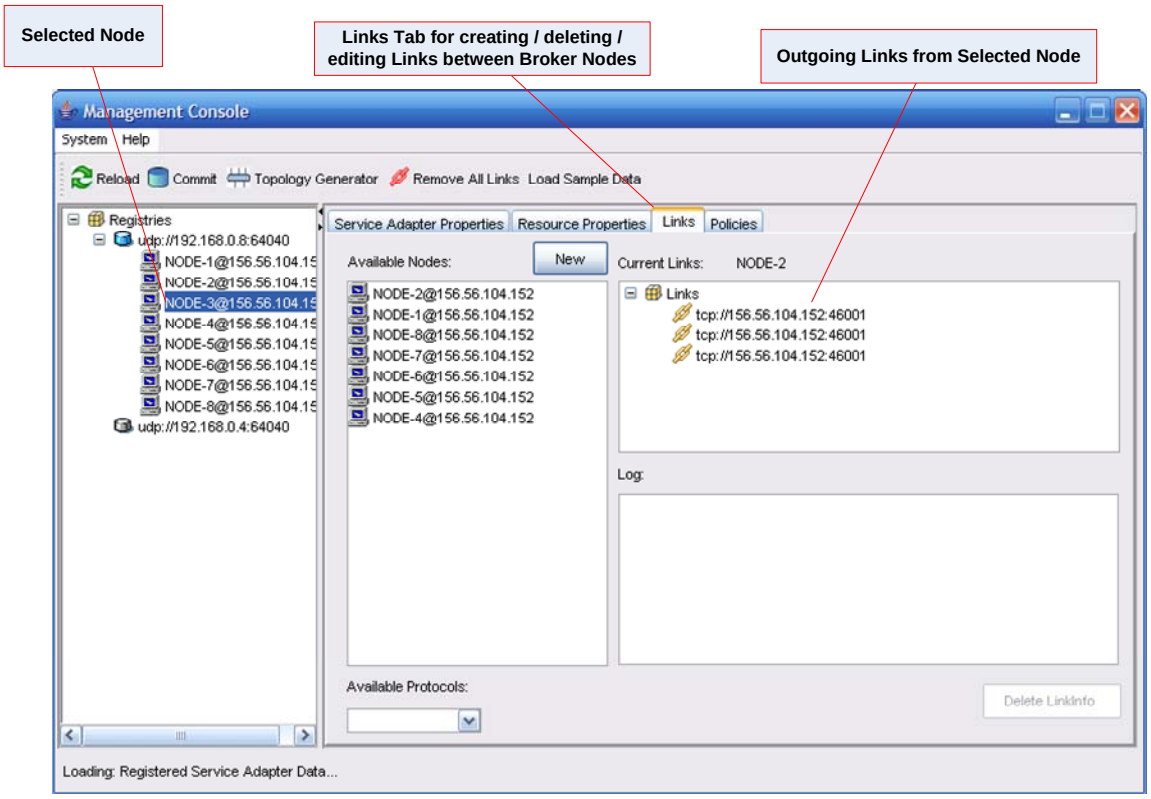


Figure A12 Editing Links

5. Editing Links

The *Links* tab allows a user to edit pre-created links (via the topology generator) or create / delete / modify user-defined links. Figure A12 shows the links created in an earlier run of CLUSTER topology generator. **`NODE-3`** has 3 out-going links.

Deleting Existing Links

To delete a link, simply select the link to delete and click on *Delete LinkInfo* as shown in Figure A13.

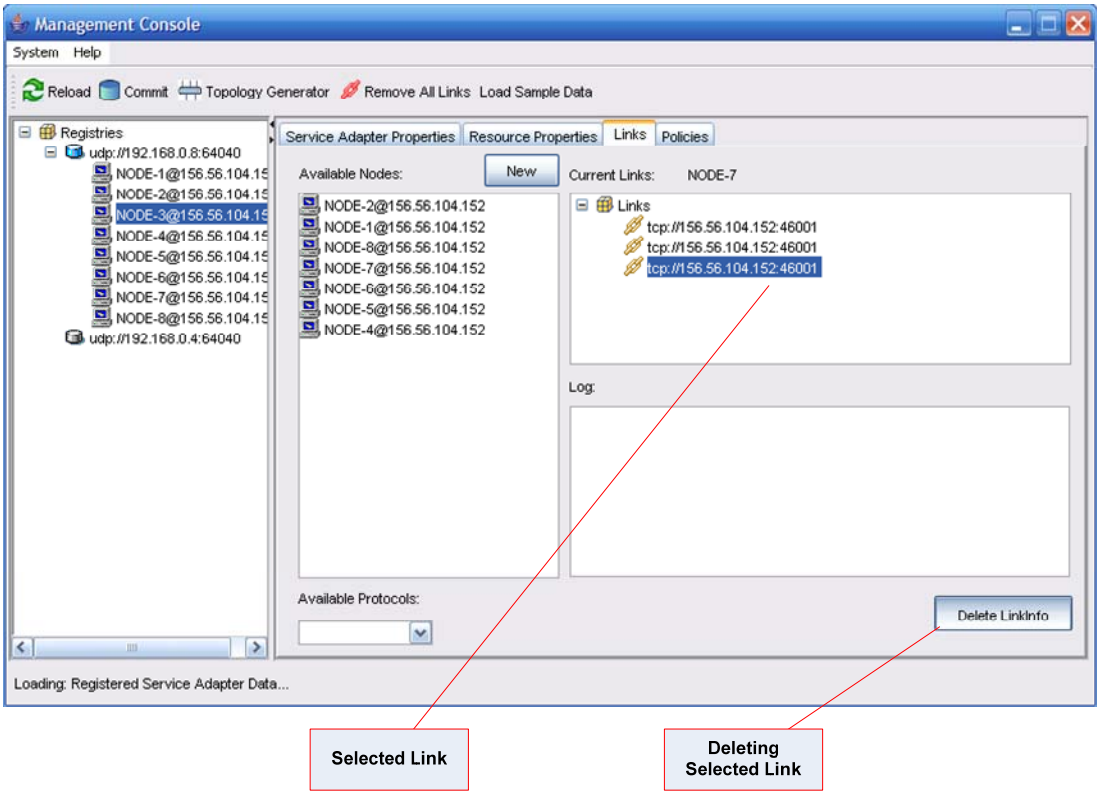


Figure A13 Deleting Links

Manually Creating Links

While creating links, the following must be noted

Links can only be created between configured nodes. i.e. nodes which have been assigned properties in the *Resource Properties* tab after creating the node via *Create Node*.

If the configuration changes after creating links, then the created links may not be deployed properly. This is because the link information contains physical IP addresses and port of the destination broker and this information is set when the link information is created. Thus, it is necessary to first set the broker configuration and then create the link.

A link using a specific protocol between 2 nodes can only be created once and is directional, i.e. if a **`TCP`** link exists from **`NODE-1`** to **`NODE-2`**, another **`TCP`** link from **`NODE-1`** to **`NODE-2`** cannot be

created, however, a **TCP** link from **NODE-2** to **NODE-1** can be created. Similarly an **NIOTCP** link between **NODE-1** to **NODE-2** can be created even if a **TCP** link was previously created.

The link creation is illustrated in Figure A14.

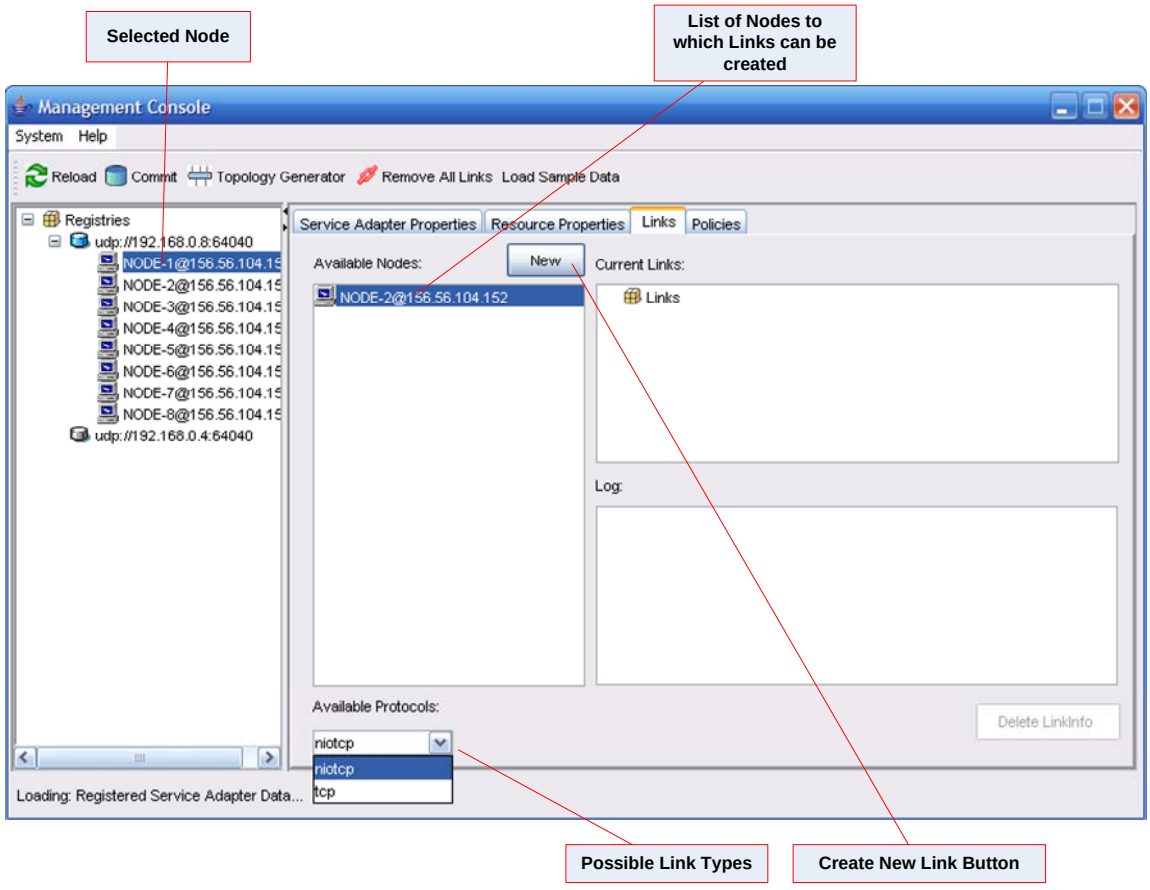


Figure A14 Manual Link Creation

To create a link from say **NODE-1** to **NODE-2**, select **NODE-1** in the left pane. The Right pane's Links tab shows the available nodes. Available nodes are instantiated nodes as defined above. Selecting an available node populates the available protocol list depending on the services configured on **NODE-2**. After selecting a protocol, simply click on the *New* button to create the link information for the link.

After nodes have been created and configured and the required link information set, the entire configuration information can be committed to registry by clicking the Commit button on the

toolbar. The manager process associated with the nodes then picks up the configuration and deploys the network of brokers as defined by the user.

6. Shutting down the Broker Network

To shutdown the broker network, simply go to the *Resource Properties* tab and click *Delete Node*. After the required nodes have been deleted, this information is committed to the registry by clicking the *Commit* button. The delete request is then acted upon by the respective manager processes.

APPENDIX – B: EXPERIMENTAL RESULTS (RUNTIME RESPONSE COST)

1. Runtime Response Cost (Single Machine):

Resources	50	100	150	200	210	220	300
	384	720	1050	1338	1568	3452	10103
	377	728	1063	1349	1541	5103	9694
	358	730	1017	1361	1513	4288	10213
	356	687	1011	1358	1517	4602	9890
	378	701	1003	1348	1559	4092	10878
	357	693	1029	1329	1489	5142	10940
	349	667	1061	1366	1404	4909	10389
	350	664	1009	1384	1390	3865	10130
	399	698	1014	1361	1411	4039	11005
	355	704	1013	1339	1465	5507	10105
Mean	366	699	1027	1353	1486	4500	10335
Standard Deviation	17	23	23	16	66	658	458
Standard Error	5	7	7	5	21	208	145

Table B 1 Response Cost (1 Manager on 1 machine)

Resources	100	200	300	400	500
	538	1079	1594	2254	6712
	546	1095	1575	2157	6414
	561	1075	1542	2181	7281
	556	1031	1522	2379	7448
	565	1043	1519	2218	8202
	539	1027	1566	2136	6261
	539	1055	1586	2237	7820
	537	1084	1491	2240	7016
	519	1059	1481	2033	6948
	536	1052	1473	2305	7277
Mean	544	1060	1535	2214	7138
Standard Deviation	14	23	44	95	602
Standard Error	4	7	14	30	190

Table B 2 Response Cost (2 Managers on 1 Machine)

Resources	100	200	300	400	500
	698	1049	1677	2245	2668
	597	1070	1592	2264	2626
	580	1113	1635	2193	2571
	579	1107	1590	2189	2774
	594	1049	1571	2152	2764
	577	1139	1529	2127	2591
	593	1062	1615	2071	2758
	598	1040	1535	2071	2711
	655	1030	1583	2075	2575
	584	1106	1538	2086	2540
Mean	606	1077	1587	2147	2658
Standard Deviation	40	37	47	73	89
Standard Error	12	12	15	23	28

Table B 3 Response Time (4 Managers on 1 Machine)

2. Runtime Response Cost (Multiple Machines):

Resources	100	200	300	400	500
	382	744	1110	1538	4739
	391	737	1064	1502	5961
	391	737	1122	1476	5052
	353	674	1053	1624	6070
	348	695	1024	1645	6067
	366	669	1139	1608	6722
	380	670	1109	1446	5492
	359	721	1091	1650	6360
	365	703	1020	1385	6340
	357	720	1023	1387	5686
Mean	369	707	1076	1526	5849
Standard Deviation	16	29	45	103	614
Standard Error	5	9	14	32	194

Table B 4 Response Time (2 Managers on 2 machines, 1 on each machine)

Resources	100	200	300	400	500
	283	501	884	764	1091
	301	445	825	858	954
	228	470	887	840	1155
	311	439	622	879	1101
	233	522	781	821	905
	307	503	728	837	926
	207	483	837	710	1140
	252	470	797	815	1087
	249	429	781	853	1191
	298	460	720	729	1272
Mean	267	472	786	811	1082
Standard Deviation	38	30	81	57	120
Standard Error	12	10	26	18	38

Table B 5 Response Time (4 Managers on 4 Machines, 1 on each machine)

APPENDIX – C: GLOSSARY OF TERMS USED

1. **D (Maximum requests that can be handled by a manager process)**

   Refers to the maximum number of requests that can be managed by a single Manager Process. For analysis sake, we assume this to be the number of resources assigned to a manager process

2. **Manager**

   A multithreaded process that simultaneously manages multiple resources. This process comprises of multiple resource manager threads each of which manages a single resource.

3. **Resource Manager**

   A single thread of execution that contains enough logic to completely manage the resource in question.

4. **Managee**

   Refers to the resource being managed.

5. **Service Adapter**

   This is a wrapper over the Managee that provides a transport interface with the messaging node while also hosting a WS Management based message processor.

6. **Bootstrap Node**

   Refers to the bootstrapping agent that periodically runs a health check manager. This component helps in maintaining fault-tolerance and ensures scalability via a hierarchical arrangement.

7. **Messaging Node**

A NaradaBrokering broker instance that forms the messaging substrate for communication between distributed components.

8. **Registry**

   A distributed database / service that persistently maintains system state.

9. **Fork Process**

   An agent that may be remotely contacted to fork off a process on the local node.

APPENDIX – D: MANAGEMENT OF RESOURCES USING WS MANAGEMENT FRAMEWORK

1. Wrapping Resources with WS Management Processor (WSManProcessor)

```
public class ResourceServiceAdapter extends WSManProcessor {
   /* Creates the Resource To manage */
   ResourceToManage resource = new ResourceToManage();

   /* Initialize the WS Management interface.
      This is done through the Service Adapter */
   public ResourceServiceAdapter(MessageSender messageSender,
            String universalEndpoint) {
      super(messageSender, universalEndpoint);
   }

/* The resource exports a GET operation that returns the time */
   public void processWxfGet(EnvelopeDocument envelopeDocument,
            MessageHeaders headers, XmlFragmentDocument xmlFrag)
            throws WSManServiceException {
      String x = resource.getTime();
      XmlObject resp = XmlObject.Factory.newValue(x);
      sendMessage(WSManActionURIs.Wxf_GetResponse, headers, resp);
   }

   /* The PUT operation is NOT supported, so throw a fault */
   public void processWxfPut(EnvelopeDocument envelopeDocument,
            MessageHeaders headers, XmlFragmentDocument xmlFrag)
            throws WSManServiceException {
      log.warn("wsman:UnsupportedFeature");

      // wsman:UnsupportedFeature :: wsman:faultDetail/NotSupported
      FaultDetailDocument fdd = WSManServiceUtil.
                  GenerateFaultDetailDocument(
                              WSManFaults.faultDetailNotSupported);

      throw new WSManServiceException(
                        WSManFaults.wsmanUnsupportedFeature, fdd);
   }
```

```
  /* Other operations may be similarly defined */
  . . .
}
```

2. Implementing Eventing (WSEvProcessor)

```
public class EvGenService extends WSManProcessor {

   static Logger log = Logger.getLogger("EvGenService");

   public EvGenService(MessageSender ms, String endpoint) {
      super(ms, endpoint);
      DummyEventGenerator evGen = new DummyEventGenerator(this,
               SubscriptionManagerEndpoint.subManagerULoc);
      setWSEvProcessor(evGen);
   }

   // Other WSManProcessor operations...
}

public class DummyEventGenerator
      extends WSEvProcessor implements Runnable {

   static Logger log = Logger.getLogger("DummyEventGenerator");

   /**
    * @param manProc
    * @param subscriptionManagerEndpoint
    */
   public DummyEventGenerator(WSManProcessor manProc,
            UniversalLocator subscriptionManagerEndpoint) {
      super(manProc, subscriptionManagerEndpoint.toString());
      new Thread(this).start();
   }

   public void run() {


      // Generate an event every 10 seconds...
      while (true) {
         try {
            Thread.sleep(10000);
         } catch (InterruptedException e) {
```

```
            log.error("", e);
         }

         String event = "<Time><SystemTime>"
                        + System.currentTimeMillis()
                        + "</SystemTime></Time>";

         System.out.println("EVENT: " + event);

         XmlObject body = null;
         try {
            body = XmlObject.Factory.parse(event);
         } catch (XmlException e) {
            log.error("", e);
         }

         try {
            // Sends the event to topic “/test”
            sendEvent(body, null, "/test");
         } catch (Exception e) {
            log.error("Error sending message ! Discarding...", e);
         }
      }
   }
}
```

3. Enumerating set of values (WSEnProcessor)

```
public class EnumService extends WSManProcessor {

   static Logger log = Logger.getLogger("EnumService");

   /**
    * @param transportType
    * @param param
    * @param bufferServicePort
    */
   public EnumService(MessageSender ms, int transportType,
                      String param, int bufferServicePort) {
      // , String endpoint) {
      super(ms, "");
      UUIDEnumerator enumService = new UUIDEnumerator(this);
      setWSEnProcessor(enumService);
```

```
   }

   // Other WSManProcessor operations...
}

public class UUIDEnumerator extends WSEnProcessor {

   // Stores whats being enumerated
   private Hashtable table;

   private Hashtable enumStatus;

   public UUIDEnumerator(WSManProcessor manProc) {
      super(manProc);
      table = new Hashtable();
      enumStatus = new Hashtable();
   }

   public EnumerationInfo Enumerate(MessageHeaders headers,
                                    Enumerate enumerate)
         throws WSManServiceException {
      String context = UIDGenerator.getUUID();
      Object o = enumerate.getExpires();

      Calendar now = new GregorianCalendar();
      GDuration expiry;

      if (o instanceof GDuration) {
         // A duration is specified,
         // add it to now to get the expiry time
         GDuration dur = (GDuration) o;
         GDuration now_Dur = XmlDateFactory.CalendarToGDuration(now);
         expiry = now_Dur.add(dur);
      } else {
         XmlDateTime xdt = (XmlDateTime) o;
         expiry =
            XmlDateFactory.CalendarToGDuration(
                  xdt.getCalendarValue());
      }

      // IF the expiry gets set to some value before now, then throw an
      // exception
      if (XmlDateFactory.GDurationToCalendar(expiry).before(now)) {
            throw new WSManServiceException(
               WSManFaults.wsenInvalidExpirationTime, null);
```

```
   }

   EnumerationInfo eInfo = new EnumerationInfo(context,
            enumerate.getEndTo(),
            XmlDateFactory.GDurationToCalendar(expiry), enumerate
                     .getFilter());

   table.put(context, expiry);

   // Lets say for example sake, that out Enuemrator returns
   // max of 10 elements before giving out an <wsen:EndOfSequence>
   enumStatus.put(context, new Integer(10));

   return eInfo;
}

public GetStatusResponseDocument GetStatus(MessageHeaders headers,
         String context, GetStatusDocument getStatusDoc)
         throws WSManServiceException {

   GDuration gd = (GDuration) table.get(context);

   GetStatusResponseDocument respDoc =
         GetStatusResponseDocument.Factory.newInstance();
   GetStatusResponse resp = respDoc.addNewGetStatusResponse();

   resp.setExpires(gd);
   return respDoc;
}

public PullResponseDocument Pull(MessageHeaders headers,
         String context, PullDocument pullDoc)
               throws WSManServiceException {

   // NOTE: CONTEXT VALIDATION HAS BEEN PREVIOUSLY DONE !
   PullResponseDocument pullRespDoc = PullResponseDocument.Factory
            .newInstance();
   PullResponse pullResp = pullRespDoc.addNewPullResponse();

   pullResp.setEnumerationContext(WSEnServiceUtil
            .ObjectToEnumerationContext(context));

   ItemListType listType = ItemListType.Factory.newInstance();

   int maxElems = pullDoc.getPull().getMaxElements().intValue();
```

```
   int elementsRemaining =
         ((Integer) enumStatus.get(context)).intValue();

   for (int i = 0; i < maxElems; i++) {

      if (elementsRemaining == 0) break;

      // Generate / Get the next element to send
      String element = "[" + elementsRemaining + "]: "
               + UIDGenerator.getUUID();
      UUIDEntryDocument uuidDoc =
               UUIDEntryDocument.Factory.newInstance();
      UUIDEntry entry = uuidDoc.addNewUUIDEntry();

      entry.setEntryNumber(i);
      entry.setUUID(element);
      XmlContentTransfer.copyAsLastChild(uuidDoc, listType);
      --elementsRemaining;
   }

   if (elementsRemaining == 0) {
      pullResp.addNewEndOfSequence();
      enumStatus.remove(context);
      table.remove(context);
   } else {
      enumStatus.put(context, new Integer(elementsRemaining));
   }

   pullResp.setItems(listType);

   return pullRespDoc;
}

public void Release(MessageHeaders headers, String context,
         ReleaseDocument releaseDoc) throws WSManServiceException {

   // OK, simply clear off the resources
   enumStatus.remove(context);
   table.remove(context);
}
public RenewResponseDocument Renew(MessageHeaders headers,
         String context, RenewDocument renewDoc)
               throws WSManServiceException {
   // We do not allow renewals, so simply throw a fault
```

```
      throw new WSManServiceException(
            WSManFaults.wsenUnableToRenew, null);
   }
}
```

4. Custom Operations

```
public class ResourceSpecificOperations
      extends ResourceSpecificOperationProcessor {

   public CustomOperations(WSManProcessor wsManProcessor) {
      super(wsManProcessor);
   }

   /* Provides Custom Operation, Init UUID,
      Action: http://test.com/InitUUID    */
   public void processResourceSpecificMgmtRequest(
            EnvelopeDocument envelopeDocument, MessageHeaders headers,
            XmlFragmentDocument xmlFrag) throws WSManServiceException {

      if (headers.xGetAction().equals("http://test.com/InitUUID")) {
            // Does some resource-specific operation...
            invokeUUIDInitProc();
            wsManagementProcessor.sendMessage(
                  "http://test.com/InitUUIDResponse", headers, null);
      }
      else {
         // Throw an operation not supported exception
         log.warn("wsman:UnsupportedFeature");

         // wsman:UnsupportedFeature :: wsman:faultDetail/NotSupported
         FaultDetailDocument fdd =
            WSManServiceUtil.GenerateFaultDetailDocument(
                  WSManFaults.faultDetailNotSupported);
         throw new WSManServiceException(
                  WSManFaults.wsmanUnsupportedFeature, fdd);
      }
   }

   private void invokeUUIDInitProc() {
      ...
   }
}
```

REFERENCES

[1] Channabasavaiah, K., K. Holley, and J. Edward Tuggle. *Migrating to a Service Oriented Architecture*. Dec 2003, Available from: http://www-128.ibm.com/developerworks/library/ws-migratesoa/.

[2] Channabasavaiah, K., K. Holley, and J. Edward Tuggle. *Migrating to a Service Oriented Architecture - Part 2*. Dec 2003, Available from: http://www-128.ibm.com/developerworks/library/ws-migratesoa2/.

[3] Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. *Web Services Architecture*. feb 11, 2004, Available from: http://www.w3.org/TR/ws-arch/.

[4] Case, J., M. Fedor, M. Schoffstall, and J. Davin. *A Simple Network Management Protocol (SNMP)*. 1990, Available from: RFC: 1157, http://www.ietf.org/rfc/rfc1157.txt.

[5] Kreger, H., *Java Management Extensions for application management*. IBM Systems Journal, 2001. **40**(1).

[6] Distributed Management Task Force, I. *Common Information Model (CIM)*. Available from: http://www.dmtf.org/standards/cim/.

[7] Distributed Management Task Force, I. *Web-Based Enterprise Management (WBEM)*. Available from: http://www.dmtf.org/standards/cim/.

[8] Czajkowski, K., D. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. *The WS-Resource Framework*. May 2004.

[9] Silberschatz, A. and P.B. Galvin, *Operating Systems Concepts*. Fifth Edition ed. 1999: Addison Wesley Longman, Inc.

[10] *Condor Project*. Available from: http://www.cs.wisc.edu/condor/.

[11] *Grid Resource Allocation Manager*. Available from: http://www.globus.org/toolkit/docs/3.2/gram/ws/index.html.

[12] Oppenheimer, D., A. Ganapathi, and D.A. Patterson. *Why do Internet services fail, and what can be done about it ?* in *USENIX Symposium on Internet Technologies and Systems (USITS '03)*. March 2003.

[13] Kephart, J.O. and D.M. Chess, *The Vision of Autonomic Computing*. IEEE Computer Magazine, 2003. **36**(1): p. 41-50.

[14] Research, I. *Research Projects in Autonomic Computing*. 2003, Available from: http://www.research.ibm.com/autonomic/research/projects.html.

[15] *LCG - LHC Computing Grid Project*. Available from: http://lcg.web.cern.ch/LCG/.

[16] *Amazon Web Services*. Available from: http://aws.amazon.com.

[17] Microsoft. *Windows Management Instrumentation (WMI)*. Available from: http://www.microsoft.com/whdc/system/pnppwr/wmi/default.mspx.

[18] Aydin, G., M.S. Aktas, G.C. Fox, H. Gadgil, M. Pierce, and A. Sayar. *SERVOGrid Complexity Computational Environments (CCE) Integrated Performance Analysis*. in *6th IEEE/ACM International Workshop on Grid Computing Grid2005 Conference*. Nov 2005. Seattle, WA: p. 256 - 261.

[19] *HPSearch*. Available from: http://www.hpsearch.org/.

[20] Aktas, M.S., G.C. Fox, and M. Pierce, *Fault Tolerant High Performance Information Services for Dynamic Collections of Grid and Web Services.* (To Appear) in "*Semantic Grid and Knowledge Grid: Systems and Applications*" Special Issue of Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Models and Applications.

[21] Vretanos, P. *Web Feature Service Implementation Specification*. 2002, Available from: OpenGIS project Document: OGC 02-058, version 1.0.0, http://www.opengeospatial.org/standards/wfs.

[22] Beaujardiere, J.d.L. *Web Map Service*. 2004, Available from: OpenGIS project Document: OGC 04-024, http://www.opengeospatial.org/standards/wms.

[23] *Global MultiMedia Conferencing System (GlobalMMCS)*. Available from: Project page: http://www.globalmmcs.org.

[24] Tanenbaum, A.S. and M.v. Steen, *Distributed Systems: Principles and Paradigms*. 1st edition ed: Prentice Hall.

[25] Lamport, L., *Time, Clocks, and the Ordering of Events in a Distributed System.* ACM Communications, July 1978. **21**(7): p. 558-565.

[26] Rodrigues, L., H. Fonseca, and P. Verissimo. *Totally Ordered Multicast in Large-Scale Systems*. in *16th Intl. Conf. on Distributed Computing Systems*. 1996: p. 503-510.

[27] *Oracle*. Available from: http://www.oracle.com.

[28] Foster, I., N.R. Jennings, and C. Kesselman. *Brain Meets Brawn: Why Grid and Agents Need Each Other*. in *3rd International Joint Conference on Autonomous Agents and Multi Agent Systems, AAMAS'04*. July 2004. New York, NY: p. 8 - 15.

[29] Nguyen, X.T. and R. Kowalczyk. *Enabling Agent-Based Management of Web Services with WS2Jade*. in *5th International Conference on Quality Software (QSIC)*. 2005.

[30] Eddon, G. and H. Eddon. *Understanding the DCOM Wire Protocol by Analyzing Network Data Packets*. March 1998, Available from: http://www.microsoft.com/msj/0398/dcom.aspx.

[31] *The Object Management Group (OMG)*. . Available from: http://www.omg.org/technology/documents/.

[32] Javasoft. *Java Remote Method Invocation - Distributed Computing for Java (White Paper)*. 1999, Available from: http://java.sun.com/marketing/collaterral/javarmi.html.

[33] Suri, N., J.M. Bradshaw, M.R. Breedy, P.T. Groth, G.A. Hill, R. Jeffers, T.S. Mitrovich, B.R. Pouliot, and D.S. Smith. *NOMADS: toward a Strong and Safe Mobile Agent System*. in *4th International Conference on Autonomous Agents*. 2000. Barcelona, Spain: p. 163-164.

[34] Garbacki, P., B. Biskupski, and H. Bal. *Transparent Fault Tolerance for Grid Applications*. in *European Grid Conference (EGC2005)*. Feb 2005. Amsterdam, The Netherlands.

[35] *XCAT Project at Indiana University*. Available from: http://www.extreme.indiana.edu/xcat/.

[36] *OpenMPI*. Available from: http://www.openmpi.org.

[37] *FT-MPI*. Available from: http://icl.cs.utk.edu/ftmpi/.

[38] Al-Tawil, K.M., M. Bozyigit, and S.K. Naseer. *A Process Migration Subsystem for a Workstation-Based Distributed Systems* in *5th IEEE*

International Symposium on High Performance Distributed Computing (HPDC-5 '96). 1996. Los Alamitos, CA.

[39] Chandy, K.M. and L. Lamport, *Distributed snapshots: Determining global states of distributed systems.* ACM Transactions on Computer Systems, Feb 1985. **3**(1): p. 63-75.

[40] Elnozahy, M., L. Alvisi, Y.-M. Wang, and D.B. Johnson. *A Survey of Rollback-Recovery Protocols in Message-Passing Systems, Technical Report (CMU-CS-99-148), School of Computer Science, Carnegie Mellon University*. June 1999.

[41] Sankaran, S., J.M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, *The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing.* International Journal of High Performance Computing Applications, 2005. **19**(4): p. 479-493.

[42] Krishnan, S. and D. Gannon. *Checkpoint and Restart for Distributed Components in XCAT3*. in *5th IEEE/ACM International Workshop on Grid Computing (Grid 2004)*. Nov 2004.

[43] Bronevetsky, G., D. Marques, K. Pingali, and P. Stodghill, *Automated application-level checkpointing of mpi programs.* Principles and Pratice of Parallel Programming, June 2003.

[44] Mockapetris, P. *Domain Names - Implementation and Specification*. Nov 1987, Available from: RFC: http://tools.ietf.org/html/rfc1035.

[45] Newman, H.B., I.C. Legrand, P. Glavez, P. Voicu, and C. Cirstoiu. *MonALISA: A Distributed Monitoring Services Architecture*. in *CHEP 2003*. March 2003. La Jola, CA.

[46] Renesse, R.V., K.P. Birman, and W. Vogels, *Astrolabe: A robust and scalable technlolgy for distributed system monitoring, management and data mining.* ACM Transactions on Computer Systems, 2003. **21**(2): p. 164 - 206.

[47] Warrier, U., L. Besaw, L. LaBarre, and B. Handspicker. *The Common Management Information Services and Protocols for the Internet (CMOT and CMIP)*. 1990, Available from: http://www.ietf.org/rfc/rfc1189.txt.

[48] Massie, M., B. Chun, and D. Culler, *The Ganglia Distributed Monitoring System: Design, Implementation and Experience.* Parallel Computing, July 2004. **30**(7).

[49] Wolski, R. *Forecasting Network Performance to Support Dynamic Scheduling using the Network Weather Service*. in *High Performance Distributed Computing (HPDC)*. 1997: p. 316 - 325.

[50] BEA, CISCO, HP, IBM, and Oracle. *JSR 262: Web Services Connector for Java Management Extensions (JMX) Agents*. 2006, Available from: http://jcp.org/en/jsr/detail?id=262.

[51] IBM, HP, CA, and Cisco. *Proposal for a CIM mapping to WSDM*. 2005, Available from: ftp://www6.software.ibm.com/software/developer/library/ws-wsdm.pdf.

[52] Arora, A., J. Cohen, J. Davis, M. Dutch, and et.al. *Web Services for Management*. June 2005, Available from: https://wiseman.dev.java.net/specs/2005/06/management.pdf.

[53] HP. *Web Services Distributed Management (WSDM)*. March 2005, Available from: http://devresource.hp.com/drc/specifications/wsdm/index.jsp.

[54] OASIS-TC. *Web Services Distributed Management: Management Using Web Service (MUWS 1.0) Part 1 & 2, OASIS Standard*. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm.

[55] OASIS-TC. *Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.0 OASIS Standard*. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm.

[56] WebMethods and HP. *Open Management Interface*. Available from: http://www1.webmethods.com/PDF/OMI_Spec.pdf.

[57] HP, IBM, Intel, and Microsoft. *Toward Converging Web Service Standards for Resources, Events, and Management*. Available from: http://msdn.microsoft.com/library/en-us/dnwebsrv/html/convergence.asp.

[58] Eugster, P.T., P.A. Felber, R. Guerraoui, and A.-M. Kermarrec, *The many faces of publish/subscribe*. ACM Computing Surveys (CSUR), June 2003. **35**(2): p. 114-131.

[59] Strom, R., G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. *Gryphon: An Information Flow Based Approach to*

Message Brokering. in *International Symposium on Software Reliability Engineering*. 1998.

[60] Carzaniga, A., D.S. Rosenblum, and A.L. Wolf. *Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service*. in *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000)*. July 2000. Portland, OR.

[61] Segal, B. and D. Arnold. *Elvin has left the building: A publish/subscribe notification service with quenching*. in *AUUG97*. Sep 1997. Brisbane, Australia: p. 243 - 255.

[62] *NaradaBrokering: Project Web Site*. Available from: http://www.naradabrokering.org.

[63] Pallickara, S. and G. Fox. *NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids*. in *ACM/IFIP/USENIX International Middleware Conference*. 2003.

[64] *ServoGrid*. Available from: http://www.servogrid.org.

[65] *Anabas, Inc.*, Available from: http://www.anabas.com.

[66] Pallickara, S. and G. Fox. *A Scheme for Reliable Delivery of Events in Distributed Middleware Systems*. in *IEEE International Conference onf Autonomic Computing*. New York, NY: p. 328-329.

[67] Pallickara, S., M. Pierce, H. Gadgil, G. Fox, Y. Yan, and Y. Huang. *A Framwork for Secure End-to-End Delivery of Messages in Publish / Subscribe Systems*. in *7th IEEE/ACM International Conference on Grid Computing (Grid 2006)*. 2006. Barcelona, Spain.

[68] *Java Message Service*. Available from: http://java.sun.com/products/jms.

[69] Pallickara, S., G. Fox, B. Yildiz, and S.L. Pallickara. *On the Costs of Reliable Messaging in Web/Grid Service Environments*. in *IEEE International Conference on e-Science & Grid Computing*. 2005. Melbourne, Australia: p. 344 - 351.

[70] BEA, Microsoft, IBM, and T. Software. *Web Services Reliable Messaging Protocol (WS-ReliableMessaging)*. March 2004, Available from: ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200403.pdf.

[71] *Web Services Reliable Messaging TC WS-Reliability*. Working draft (Jan 26, 2004), Available from: http://www.oasis-

open.org/committees/download.php/5155/WS-Reliability-2004-01-26.pdf.

[72] Microsoft, IBM, and BEA. *Web Services Eventing (WS – Eventing)*. Aug 2004, Available from: http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf.

[73] Pallickara, S., G. Fox, M. Aktas, H. Gadgil, B. Yildiz, S. Oh, S. Patel, M. Pierce, and D. Yemme. *A Retrospective on the Development of Web Service Specifications*. July 2006.

[74] Gadgil, H., G. Fox, S. Pallickara, and M. Pierce. *Managing Grid Messaging Middleware*. in *Challenges of Large Applications in Distributed Environments (CLADE)*. 2006. Paris, France: p. 83 - 91.

[75] *Fault Tolerant High Performance Information Service, http://www.opengrids.org/extendeduddi/*.

[76] Pallickara, S., H. Gadgil, and G. Fox. *On the Discovery of Brokers in Distributed Messaging Infrastructures*. in *IEEE Cluster*. Sep 27 - 30, 2005. Boston, MA.

[77] Bunting, B., M. Chapman, O. Hurley, M. Little, J. Mischinkinky, E. Newcomer, J. Weber, and K. Swenson. *Web Services Context (WS-Context)*. Available from: http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf.

[78] Pallickara, S., G. Fox, and H. Gadgil. *On the Discovery of Topics in Distributed Publish/Subscribe systems*. in *6th IEEE/ACM International Workshop on Grid Computing Grid 2005*. 2005. Seattle, WA: p. 25-32.

[79] Microsoft and BEA. *Web Service Enumeration (WS - Enumeration)*. Sep 2004, Available from: http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-enumeration.pdf.

[80] Microsoft. *Web Service Transfer (WS - Transfer)*. Sep 2004, Available from: http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-transfer.pdf.

[81] *Apache XML Beans*. Available from: http://xmlbeans.apache.org.

[82] *Network Topologies*. Available from: http://en.wikipedia.org/wiki/Network_topology.

[83] Gunduz, G., S. Pallickara, and G. Fox, *An Efficient Scheme for Aggregation and Presentation of Network Performance in Distributed Brokering Systems.* Systemics, Cybernetics and Informatics, 2004.

[84] Stoica, I., R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications*. in *SIGCOMM*. 2001.

[85] Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Schenker. *A scalable content-addressable network*. in *SIGCOMM*. 2001.

[86] Uyar, A., *Scalable Service Oriented Architecture for Audio/Video Conferencing*. 2005, Syracuse University.

[87] Rosenberg, J., J. Weinberger, C. Huitema, and R. Mahy. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. 2003, Available from: http://www.ietf.org/rfc/rfc3489.txt.

[88] Rosenberg, J., R. Mahy, and C. Huitema. *TURN: Traversal using Relay NAT*. July 2004.

[89] Guha, S. and P. Francis. *Characterization and Measurement of TCP Traversal through NATs and Firewalls*. in *Internet Measurement Conference (IMC)*. Oct 2005. Berkeley, CA.

[90] Ford, B., P. Srisuresh, and D. Kegel. *Peer-to-Peer Communication Across Network Address Translators*. 2005, Available from: http://www.brynosaurus.com/pub/net/p2pnat/.

[91] Guha, S., Y. Takeda, and P. Francis. *NUTSS: A SIP based Approach to UDP and TCP Connectivity*. in *SIGCOMM'04 Workshop*. Aug 2004. Portland, OR: p. 43 - 48.

[92] IBM, B. Systems, Microsoft, S. AG, S. Software, and VeriSign. *Web Services Policy Framework (WS - Policy)*. March 2006, Available from: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-polfram/ws-policy-2006-03-01.pdf.

[93] Aktas, M., G. Fox, and M. Pierce. *Fault Tolerant High Performance Information Services for Dynamic Collections of Grid and Web Services*. in *FGCS Special issue from 1st International Conference on SKG2005 Semantics, Knowledge and Grid*. Nov 27-29, 2005. Beijing, China.

[94] Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International J. Supercomputer Applications, 2001. **15**(3).

[95] Microsoft, IBM, and VeriSign. *Web Services Security (WS-Security) Version 1.0*. April 2002, Available from: http://www.verisign.com/wss/wss.pdf.

Vita

Name:	Harshawardhan Gadgil
Date of Birth:	May 24, 1979
Place of Birth:	Mumbai, India

Education:

May, 2002	M.S., Computer Science Indiana University, Bloomington, Indiana
May, 2000	B.E., Computer Engineering Mumbai University, Mumbai, India

Experience:

Jan 2003 ~ Aug 2006	Research Assistant Community Grids Lab, Indiana University Bloomington, Indiana
May 2001 ~ July 2001	Summer Intern Microsoft Corporation, Redmond, Washington
Aug 2000 ~ December 2002	Teaching Assistant Indiana University, Bloomington, Indiana