Scalable, Fault-tolerant Management in a Service Oriented Architecture

Harshawardhan Gadgil, Geoffrey Fox, Shrideep Pallickara, Marlon Pierce
Community Grids Lab, Indiana University, Bloomington IN 47404
(hgadgil, gcf, spallick, marpierc)@indiana.edu

Abstract

The service-oriented architecture has come a long way in solving the problem of reusability of existing software resources. Grid applications today are composed of a large number of loosely coupled services. While this has opened up new avenues for building large, complex applications, it has made the management of the application components a non-trivial task. Management is further complicated when services exist on different platforms, are written in different languages, present in varying administrative domains restricted by firewalls and are susceptible to failure.

This paper investigates problems that emerge when there is a need to uniformly manage a set of distributed services. We present a scalable, fault-tolerant management framework. Our empirical evaluation shows that the architecture adds an acceptable number of additional resources making the approach feasible.

Keywords: Scalable, Fault-tolerance, Service Oriented Architecture, Web Services

1. Introduction

This Service Oriented Architecture (SOA) [1] delivers unprecedented flexibility and cost savings by promoting reuse of software components. This has opened new avenues for building large complex distributed applications by loosely coupling interacting software services. A distributed application benefits from properly managed (configured, deployed and monitored) services. However the various technologies used to deploy, configure, secure, monitor and control distributed services have evolved independently. For instance, many network devices use Simple Network Management Protocol [2], Java applications use Java Management eXtensions (JMX) [3] while servers implement management using Web-based Enterprise Management [4] or Common Information Model [5].

The Web Services community has addressed this challenge by adopting a SOA using Web Services technology to provide *flexible* and *interoperable* management protocols. The *flexibility* comes from the ability to quickly adapt to rapidly changing environments. The *interoperability* comes from the use of XML based interactions that facilitate implementations in different languages, running on different platforms and over multiple transports.

1.1. Aspects of Resource / Service Management

Before we proceed further, we clarify the use of term *Resource* in this paper. Distributed applications are composed of components which are digital entities on the network. We consider a specific case of distributed applications where such digital entities can be controlled by zero or modest state. These digital entities in turn can bootstrap and control components with much higher state. Such components could be hardware (e.g. network, CPU, memory) or software (e.g. file-systems, databases, services). We consider the combination of such a digital 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 WSRF). Further, if the digital entity is a service we add appropriate management interfaces. If the digital entity is not a service we create a Web Service wrapper that augments a Web Service based management interface to the manageable resource.

Next, we discuss the scope of management with respect to the discussion presented in this paper. The primary goal of Resour*ce Management* is the efficient and effective use of an organization's resources. Resource management can be defined as *"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 other that deals with resource administration.

In the first category, resource management provides resource allocation and scheduling, the goal being sharing resources fairly while maintaining optimal resource utilization. For example, operating systems [6] provide resource management by providing fair resource sharing via process and memory management. Condor [7] provides specialized job management for compute intensive tasks. Similarly Grid Resource

Allocation Manager [8] provides an interface for requesting and using remote system resources for job execution.

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, bootstrapping) and dynamic (monitoring and event handling) aspects.

This paper deals with the second category of resource / service management.

1.2. Motivation

As suggested by Moore's Law [9], the phenomenal progress of technology has driven the deployment of an increasing number of devices ranging from RFID devices to supercomputers. These devices are widely deployed, spanning corporate administrative domains typically protected by firewalls. Low cost of hardware has made replication a cost-effective approach to fault-tolerance especially when using software replication. 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 to successfully manage such large scale systems.

First, as the size of an application increases in terms of factors such as hardware components, software components and geographical scale, it is certain that some parts of the application will fail. An analysis [10] of causes of failure in Internet services shows that most of the services's downtime can be attributed to improper management (such as wrong configuration) while software failures come second.

Second, administration tasks are mainly performed by persons. A great deal of knowledge needed for administration tasks is not formalized and is part of administrator's know-how and experience. With the growing size and complexity of applications, the cost of administration is increasing while the difficulty of administration tasks approaches the limits of administrator's skills.

Third, different types of resources in as system require different resource specific management frameworks. As discussed before, the resource management systems have evolved independently. This complicates 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, a central management system poses problems related to scalability and vulnerability to a single point of failure.

These factors motivate the need for a distributed management infrastructure. 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. Finally the management framework must automatically handle failures within itself.

1.3. Desired Features

In this section, we provide a summary of the desired characteristics of the management framework:

Fault-tolerance: As applications span wide area networks, they become difficult to maintain and resource failure is normal. Failure could be a result of the actual resource failure or because of some related component such as network making resource inaccessible or even because of the failure of the management framework itself. While the framework must provide *self-healing* capabilities, resource failure must be handled by providing appropriate policies to detect and handle failures while avoiding inconsistencies.

Scalability: With the increase in number of manageable resource, the framework must scale to accommodate the management of additional resources. Further, additional components are typically required to provide fault-tolerance. The framework must also scale in terms of the number of these additional components.

Performance: Additional components required to support the framework increases the initialization cost. While initialization costs are acceptable, they contribute to increasing the cost of recovery from failure. Runtime events generated by resources require a finite amount of time to process. The challenge is to achieve acceptable performance in terms of recovery from failure and responsiveness to faults.

Interoperability: As previously discussed, resources exist on different platforms and may be written in different languages. While proprietary management systems such as JMX and WMI Windows Management Instrumentation [11] have been quite successful, they are not interoperable limiting their use in heterogeneous systems and platforms. The management framework must address the interoperability issue. We leverage a Web Service based management protocol to address interoperability.

Generality: Resource management must be generic, i.e. the framework must apply equally well to hardware as well as software resources. Resource specific

management would still be required while the framework guarantees the basic features such as scalability and fault-tolerance.

Usability: As explained before, large scale systems are difficult to manage especially when components fail. The management framework must be usable in terms of autonomous operation provided whenever possible.

Thus, the framework must provide self-healing properties by appropriately detecting failures and instantiating new instances of failed management framework components with minimum user interaction.

The rest of the paper is organized as follows: We describe the framework in **Section 2**. We evaluate our system in **Section 3** and discuss the feasibility of the system. A sample application is presented in **Section 4**. We present related work in **Section 5**. **Section 6** is conclusion and future work.

2. Architecture

Our approach uses intrinsically robust and scalable management services and relies only on the existence of a reliable, scalable database to store system state. The system leverages well known strategies for providing fault-tolerance such as passive replication that helps provide simplicity of implementation.

To scale the system to a wide area we use a hierarchical bootstrapping mechanism. Scaling is improved locally by using a publish/subscribe based distributed messaging systems such as NaradaBrokering **[12]** by multiplexing communication over single connection and appropriately publishing or subscribing to relevant topics. Further NaradaBrokering helps provide a transport independent communication framework that can tunnel through firewalls thus enabling resources in restricted administrative domains to be managed.

2.1. Components

The overall management framework is shown in **Figure 1**. It consists of units arranged hierarchically and controlled by a bootstrap node. The hierarchical organization of units makes the system scalable in a wide-area deployment. 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.

2.1.1. **Resource** As defined in **Section 1.1** we refer to Resource as the component that requires management. We employ a service-oriented management architecture and hence we assume that these Resources have a Web Service interface that accepts management related messages. In the case where the Resource is not a Web Service we augment the Resource with a service adapter that serves as a management service proxy. The service adapter is then responsible for exposing the managed Resource.

2.1.2. **Service Adapter** Service adapter serves as a mediator between the manager and the Resource. Service adapter is responsible for

1. Sending periodic heartbeats to the associated Manager.
2. Providing a transport neutral connection to the manager (possibly via a messaging node). If there are multiple brokers, the Service Adapter may try different Messaging nodes to connect to, should the default 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 **[13]**.
3. Hosting a service oriented messaging based management processor protocol such as WS Management (Refer **[14]** for details on our implementation). The WS – Management processor provides basic management framework and a resource wrapper is expected to provide the correct functionality (mapping WS Management messages to resource-specific actions).

Additionally the Service Adapter may provide an interface to a persistent storage to periodically store the state to recover from failures. Alternatively, recovery may be done by resource-specific manager processes as has been implemented in our prototype.

2.1.3. **Manager** A manager is a multi threaded process and can manage multiple resources at once. Typically, one resource-specific manager module thread is responsible for managing exactly one resource and is also responsible for maintaining the resource configuration. Since every resource-specific manager only deals with the state specific to the resource it is managing, it can *independently checkpoint* the runtime state of the resource to the registry. Manager processes usually maintain very little or no state as the state can be retrieved easily by either querying the resource or looking up in the registry. This makes the managers robust as they can be easily replaced on failure.

The Manager process also runs a 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.

On failure, a finite amount of time is spent in detecting failure and re-assigning management to new manager processes (Passive Replication). When no communication is received from a managed resource, the manager always verifies if it is still responsible for managing the resource before re-establishing management ownership with the resource. This helps in preventing two managers from managing the same resource.

2.1.4. **Registry** 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.

The registry may be 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 **[15]** service. Usually read operations can be directly served from an in-memory cache but writes are always written directly to the persistent store. We assume the persistent store to be distributed and replicated for performance and fault-tolerance purposes.

A *Request Processor* provides the necessary logic for invalidating managers that have not renewed within a predefined time frame, generating a unique *Instance ID* for every new instance of resource and manager and assigning resources to managers.

2.1.5. **Messaging Node** 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 Resource, present behind a firewall or a NAT router, to be managed (for e.g. connecting to the messaging node and utilizing tunneling over `HTTP/SSL` through a firewall).

One may employ multiple messaging nodes to achieve fault-tolerance as the failure of the default node automatically causes the system to try and use the next 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.

2.1.6. Bootstrap Service 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 system health check specifically checks for presence of a working messaging node, an available registry endpoint and enough number of managers to manage all registered resources**.**

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 (henceforth, 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. The child nodes send periodic heartbeats to the parent node. Failure is detected when a heartbeat is not received within a specified timeframe.

2.1.7. **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 Resource which is then appropriately set by a Manager. In some cases there would be a group of Resources 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. Dependencies in the system in such cases are set by the user while the execution of dependencies is performed by the management architecture in a fault-tolerant manner.

2.2. Consistency

While the framework handles the basic fault-tolerance and scalability issues, it still faces many consistency issues such as duplicate requests and out of order messages. This leads to a number of consistency issues such as

1. Two or more managers managing the same resource
2. Old messages reaching after new requests
3. Multiple copies of same resource

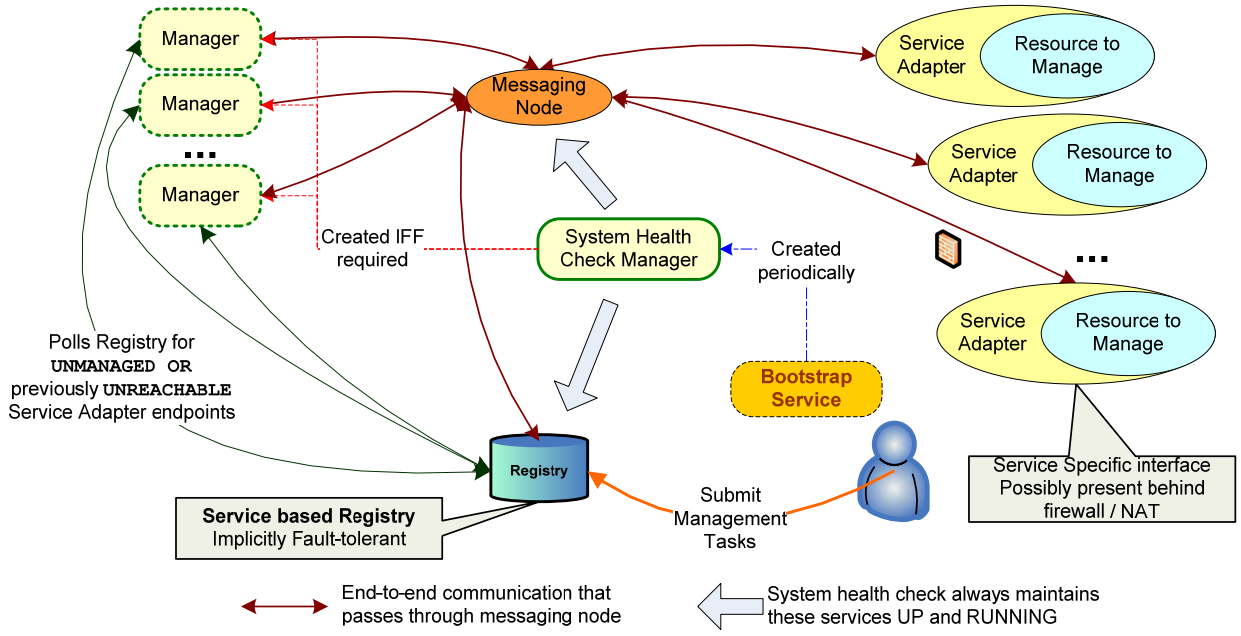

Figure 1 Overview showing components of the Management Architecture

To handle these issues, we assume the registry to generate a monotonically increasing unique instance ID (IID) for each instance of resource (managed resource or resource specific manager). Every outgoing message is tagged with a message id that comprises of the sender's instance id and a monotonically increasing sequence number. This allows us to resolve consistency issues as follows

1. Requests from manager A is considered obsolete when IID(A) < IID(B).
2. Service adapter stores the last successfully processed message's message id allowing it to distinguish between duplicates and obsoletes.

While multiple copies of same resource should be handled by policy mechanism, typically if IID (Resource 1) < IID (Resource 2) then Resource 1 is an obsolete resource so it may be detected and shutdown, thus resolving inconsistencies.

2.3. Security

A distributed system gives rise to several security issues such as but not limited to denial or service, unauthorized access to resources and modification of messages when traveling over insecure intermediaries. Although we have currently not implemented any security framework, our use of Naradabrokering allows us to leverage the security features of the substrate to cope with such attacks. The Topic Creation and Discovery mechanism [16] ensures that physical location (host and port) of any entity is never revealed. Further communication with entities can be restricted by providing access control. The security framework [17] provides end-to-end secure delivery of messages. Encrypted communication prevents unauthorized access to messages while use of digital signatures help detect possible message modifications.

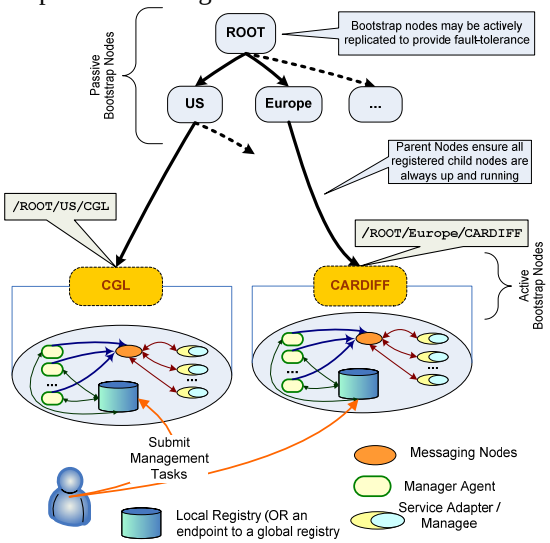

Figure 2 Achieving scalability through hierarchical arrangement

3. Performance Evaluation

In this section we analyze the system from scalability point of view. Failure recovery of framework components is currently based on timeouts as described in **Section 2.1** while correctness is handled as described in **Section 2.2**. Recovery time of managed resources is very resource specific and sample recovery costs for proof-of-concept implementation are presented in **Section 4.1**.

Our system adds a few components apart from the actual resources being managed, in order to achieve scalable, fault-tolerant management. We describe our benchmarking approach and include observed measurements. All our experiments were conducted on the Community Grids Lab's *GridFarm cluster* (GF1 – GF8). The *Gridfarm* machines consist of Dual Intel Xeon hyper-threaded CPUs (2.4 GHz), 2 GB RAM running on Linux (Linux 2.4.22-1.2199.nptlsmp). They are interconnected using a 1 Gbps network. The Java version used was Java Hotspot™ Client VM (build 1.4.2_03-b02, mixed mode). While we used JDK 1.4 for performance benchmarking, we expect better performance using JDK 1.5 or higher.

3.1. Runtime State

Our architecture uses asynchronous communication between components. Typically a domain would have one registry endpoint but the registry itself would be replicated for fault-tolerance and performance purposes. This introduces a bottleneck when performing registry read/write operations. Thus the goal is to minimize registry accesses, which in turn implies that the runtime state maintained per resource must be sufficiently small so that it can be read/written using as few a number of calls as possible.

3.2. Test Setup

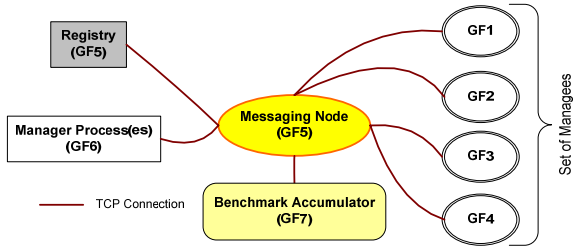

Figure 3 Test Setup

The most important factor in implementing a multi-threaded manager process is the maximum number of resources a single manager process can manage. This in turn is dependent on the response time required to handle an event from the resource. Typically the response time is resource dependent and is also affected by the actual work required in handling an event from the resource. If the handling entails one or more registry access, additional time is spent in handling the event. 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, 1 registry (possibly backed by a stable storage via WS Context service) and one or more Manager processes. Finally this number also determines how the system scales.

The test setup is shown in **Figure 3**. We ran multiple resources on the Grid cluster machines GF1 – GF4. The Messaging node and registry were run on GF5 while the Manager process was run on GF6. A benchmark accumulator process was run on GF7.

The testing methodology was as follows. The *Benchmark Accumulator* process sends a message to all the Resources and starts a timer. These Resources then generate an event and send it to their associated Manager process. The Manager process processes the event (i.e. it simply responds back to the resource with a message which corresponds to the handling of the event). Once a response is received, the resource responds back to the benchmark accumulator process. When all resources have reported, the time is noted and the difference corresponds to the overall response time. Note that this time includes an additional latency for sending the message to all resources and for all resources to respond back which is ignored considering the fact that processing time is typically much higher than latency of a message in a closed cluster of machines.

3.3. Observations

The measured response time shows a case with catastrophic failures, one in which every single resource being managed generates an event. As expected, with an increase in the number of managed resources, 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.

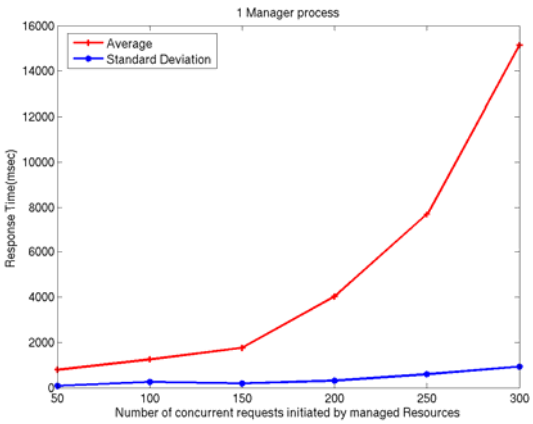

Figure 4 Average response time when handling multiple concurrent requests from different resources when using a single Manager process

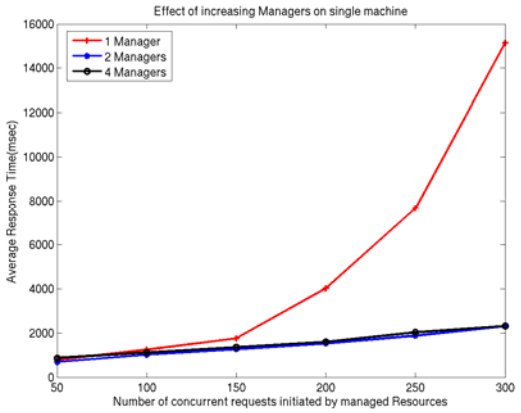

Figure 5 Average Response Time after increasing the number of manager processes on the same machine

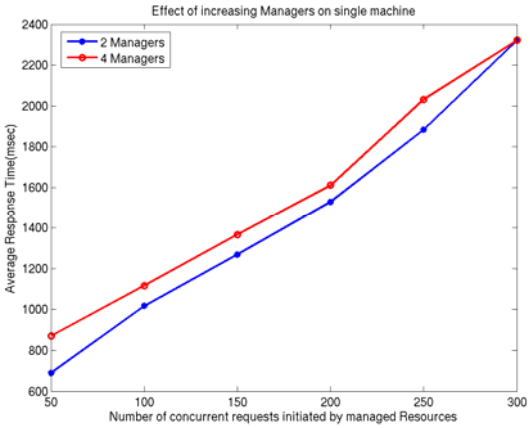

Figure 6 Average response time when handling concurrent requests using 2 and 4 manager processes on same machine

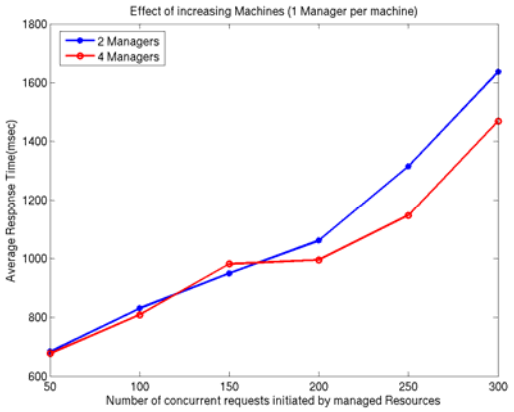

Figure 7 Average response time when handling concurrent requests using 2 and 4 manager processes on different machines

The average response time is shown in **Figure 4**. The figure shows the metrics when multiple concurrent failures are handled by a single manager process. If the number of manager processes is increased, we see a huge performance benefit by increasing the processes from 1 to 2 as shown in **Figure 5**. **Figure 6** shows the performance (close-up of **Figure 5**) when there are 2 and 4 manager processes respectively. Note that if 4 manager processes are used, instead of 2, we see that the average response time slightly increases. The reason is primarily due to the fact that our test machines had only 2 physical processors and the system takes time to context switch between various processes. We conclude that adding more manager processes than the number of available processors on a particular node, does not necessarily improve system performance especially when handling multiple failures.

Finally, we distribute the manager processes on different machines. Thus for instance, running manager processes on 2 or 4 machines instead of just 1, improves performance (comparing **Figure 7** and **Figure 4**). We note a slight gain when distributing processes over multiple machines (compare **Figure 6** and **Figure 7**). As observed from **Figure 7**, the gain (increasing machines from 2 to 4) is not very high because of the finite amount of time it takes to process each message.

3.4. Discussion of results

A single manager process can handle 1000s of requests from resources. This is because of the use of a publish/subscribe framework for communication that can multiplex requests to multiple endpoints using a single connection channel via the messaging node. However the response time when handling multiple concurrent requests increases as the number of requests increases. We note that as the number of concurrent requests increases beyond 200, the response time increases rapidly. Further, note that the number 200 is typical with respect to our test setup and could easily be higher especially in the case where the resources can tolerate higher delays. Finally, depending on the quality of service (response time) desired, the number 200 may be appropriately adjusted. For the sake of discussion in the next section, we consider 200 as an illustrative value.

Further, most resources do not require constant management (e.g. on millisecond scale) hence running more managers per processor is still acceptable. The graphs indicate a very specific case of catastrophic failure where every single resource being managed generates an event. The experiment indicates how the worst case scenario affects the average response time.

Additionally, note that the benchmark results presented here deal with only one unit of management infrastructure. We expect multiple units in a wide area deployment (Ref. **Figure 2**) to behave similarly.

3.5. Amount of Management Infrastructure Required

We now try to answer the research question, *"How much Management Infrastructure is required to handle N Resources?"* We define the term *"Management Infrastructure"* as the additional resources (processes and not physical hardware) required for providing fault-tolerant management.

Let **N** be the number of resources requiring management. If **D** is the maximum number of resources that is configured to be managed by a single manager process, then we require at-least **N/D** manager processes. 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. With 1 messaging node per leaf domain we require **N/M** leaf domains. Further, we need at least **Z/D** manager processes per leaf domain. Let **R** be the number of registry replicas used to provide fault-tolerant scalable registry. Thus total number of management infrastructure processes at the lowest leaf level is

(R registry + 1 messaging node + 1 bootstrap node + M/D managers) * (N/M such leaf domains)
= (2 + R + M/D)*N/M

In our measurements a single broker could reliably support about (**M = 800**) simultaneous TCP connections. To scale to a larger number of resources, a different protocol such as UDP may be used that improves the value of M. However, additional logic must be used to account for dropped messages via message retry and timeouts. The second approach is to use a cluster of strongly connected messaging nodes however this requires additional management in setting up links between the various messaging nodes and maintaining them in a fault-tolerant fashion. A third way is to redistribute resources such that they are in different management domains.

To manage the **N/Z** leaf domains, an additional number of passive bootstrap nodes are required. Typically the number of passive nodes would be **<< N/Z** 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, the percentage of management infrastructure required with respect to number of resources **N** is

$$MGMT_{INFRASTRUCTURE} = [(2 + R + M/D)*N/M]/N * 100\ \% = [(2+R)/M + 1/D] * 100\ \%$$

As an illustration, if **D = 200, R = 4** and **M = 800**, then $MGMT_{INFRASTRUCTURE}$ **= [(2+4)/800 + 1/200] * 100 % = 1.2 %**

Thus, as the number of resources to manage increases, fault-tolerant management of the system can be achieved by adding about **1%** more resources. Note that, when the number of resources **N** is small (e.g. **N = 10**), we still require the basic infrastructure (consisting of 1 manager, 1 bootstrap node, 1 messaging node and R registries) to manage them. Assuming **R = 4**, the minimum infrastructure components are **7**. Thus the architecture scales when **N * 1.2% = 7**, i.e. **N ≈ 600**. Thus we conclude that when **N** is large **(> 600)**, we can achieve fault-tolerant management by adding approximately **1%** additional resources.

Finally, as discussed in **Section 3.4**, the value of **D** may be suitably adjusted which would determine the number of manager processes required and the percentage of extra resources ($MGMT_{INFRASTRUCTURE}$).

4. Sample Application

The system feasibility as determined in the previous section is mainly dependent on modest run-time state maintained per resource-specific manager thread and the number of registry accesses required to retrieve / store state. One such application is management of a Grid Messaging Middleware: NaradaBrokering. The motivation for management is discussed in more detail in Ref. **[18]**. In this paper we present a discussion on application of our architecture to NaradaBrokering.

While the failures of system components are handled by the framework, resource failure handling is handled by defining resource specific policies and implemented by resource-specific managers. As a proof-of-concept, we implemented the **AUTOInstantiate** policy which automatically instantiates a broker process when the manager is unable to successfully establish contact with an existing broker's service adapter.

A broker topology determines the number of outgoing links from the managed broker. For instance consider a ring topology where each broker has one link to the next broker in the ring. In a cluster topology, each cluster has brokers connected in a ring. Each super cluster has clusters connected in a ring, while each super-super-cluster has super-clusters connected in a ring. With such a topology, the number of outgoing links range from 0 to 3 links.

4.1. Benchmarking

To find the recovery cost after failure, we benchmark the actual time it takes to create a broker from scratch after it has failed. The managed broker creates links to a static broker and we time recovery after failure. While network partitions and slowed processes would typically raise false alarms about

resource failures, we delegate the functionality of handling such situations to the resource specific manager. For our purpose, we rely on a heartbeat mechanism and timeouts to determine resource failure. The step-wise procedure followed is as follows:

1. Resource manager sends a shutdown message to broker which kills itself on receiving it.
2. Resource manager times out and starts a timer.
3. The resource manager then instantiates a copy of the failed resource which re-registers itself in the registry and the manager process (spawn remote process). The current broker state is read from registry (read state).
4. The manager then brings the resource back up (restore) to the state before failure. This includes restoring any configured outgoing links. After this, a full recovery of the failed resource has occurred and we time this recovery.

Table 1 Observed Recovery Time (msec)

Operation	Mean	Std. Dev.
Spawn Remote Process	2362	56
Read State	8	2
Restore (1 broker + 1 Link)	1420	27
Restore (1 broker + 3 links)	1615	258

The results are presented in **Table 1**. We note that a single broker resource can be recovered in about 5 sec. If there are dependencies between brokers (one managed broker connects to another managed broker), timing differences between failure detection and recovery phase could easily worsen the recovery time.

We believe that appending such management capabilities to NaradaBrokering framework is an important contribution 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 deployed system.

5. Related Work

The Web Services Resource Framework (WSRF) is a suite of specifications that align the OSGI conceptual model to be in agreement with existing Web standards. WSRF defines a *WS-Resource* as a "composition of Web Service and a stateful Resource". The WSRF defines conventions for managing state in distributed system comprising of such *WS-Resources*. The WSRF community has adopted Web Services Distributed Management (WSDM) that defines a complete management model that includes Management of Web Services (MOWS) **[19]** and Management using Web Services(MUWS) **[20]**. By contrast, we define any service that needs configuration, lifecycle and runtime management as a resource and wrap it with a service interface to expose management capabilities. Management is provided by a complementary specification, WS-Management **[21]** which is a SOAP-based protocol for managing systems (including Web Services) and functionally overlaps with MUWS. We selected WS Management primarily due to its simplicity and because we could leverage WS – Eventing **[22]** from NaradaBrokering's Web Service support.

SNMP (Simple Network Management Protocol) **[2]** deals primarily with network resources. SNMP is an application layer protocol that facilitates exchange of management information between network devices. Lack of security features however reduces SNMP to a monitoring facility only. As we have discussed in **Section 1.1**, monitoring is an important aspect of management but not all of it. There are a variety of distributed monitoring frameworks such as Ganglia **[23]**, Network Weather Service **[24]** and MonALISA **[25]**. The primary purpose of these distributed monitoring frameworks is to provide monitoring of global Grid systems and aggregation of metrics. Some systems such as MonALISA also provide the capability of configuring and managing services via RMI calls.

In the Java community, the JMX **[3]** 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 **[26]**. While JMX presents the capability to instrument applications with appropriate messages, metrics and control mechanisms, a Web Service based management protocol provides a more cross-platform, standards-based interface.

6. Conclusion and Future Work

A successful distributed application benefits from properly managed services. In this paper we have presented the need and our approach to uniformly manage a set of distributed services. To make the management framework interoperable we employed a service-oriented architecture based on WS-Management. This work leveraged the publish/subscribe paradigm to scale locally and a hierarchical distribution to scale in wide area deployments. The system is tolerant to faults within the management framework while resource failure is handled by implementing user-defined policies. When applied to resources with modest external state, the approach is feasible since it adds about 1% additional resources to provide fault-tolerant management to a large set of distributed resources.

In the future we would like to apply the framework to broader areas that would help carry out more detailed performance benchmarks tests. We believe that application of management framework to such systems can bring up many interesting research issues, specifically challenging scalability of the system. We also plan to implement Naradabrokering's security infrastructure that would help resolve the security issues as detailed in **Section 2.3**. Our current implementation uses WS – Management. In the future we would like to investigate implementing the merged **[27]** Web Service based management specifications. Finally, more metrics (such as CPU utilization, available memory and locality) need to be taken into account when assigning managers to resources.

7. References

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

[2] Case, J., et al. *A Simple Network Management Protocol (SNMP)*. 1990 RFC: 1157, http://www.ietf.org/rfc/rfc1157.txt

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

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

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

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

[7] *Condor Project*. http://www.cs.wisc.edu/condor/

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

[9] *Moore's Law*. http://en.wikipedia.org/wiki/Moore's_law

[10] 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.

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

[12] 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.

[13] 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.

[14] Pallickara, S., et al., *A Retrospective on the Development of Web Service Specifications*, Chapter in Book *Securing Web Services: Practical Usage of Standards and Specifications*, P. Panos, Editor. 2006, Idea Group Inc.: University of Newcastle Upon Tynehttp://grids.ucs.indiana.edu/ptliupages/publications/CGL-WebServices-Chapter.pdf

[15] Bunting, B., et al. *Web Services Context (WS-Context)*. http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf

[16] 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, p. 25-32, Seattle, WA.

[17] Pallickara, S., et al. *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.

[18] Gadgil, H., et al. *Managing Grid Messaging Middleware*. in *Challenges of Large Applications in Distributed Environments (CLADE)*. 2006, p. 83 - 91, Paris, France.

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

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

[21] Arora, A., et al. *Web Services for Management*. June 2005 https://wiseman.dev.java.net/specs/2005/06/management.pdf

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

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

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

[25] Newman, H.B., et al. *MonALISA: A Distributed Monitoring Services Architecture*. in *CHEP 2003*. MArch 2003, La Jola, CA.

[26] BEA, et al. *JSR 262: Web Services Connector for Java Management Extensions (JMX) Agents*. 2006 http://jcp.org/en/jsr/detail?id=262

[27] HP, et al. *Toward Converging Web Service Standards for Resources, Events, and Management*. http://msdn.microsoft.com/library/en-us/dnwebsrv/html/convergence.asp