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

With the emergence of Service-based architectures, management of an application which comprises of a large number of distributed services becomes difficult as resources appear, move and disappear across the network. As service components span different network boundaries, constraints such as network policies, firewalls and NAT devices further complicate management by limiting direct access to the resource. Services and resources may exist on different platforms and may be written in different languages, which promotes use of proprietary solutions thus affecting interoperability.

In this paper we present a novel architecture that leverages "publish-subscribe" principles for enabling scalable and fault-tolerant management of a set of distributed entities. We make management interoperable by leveraging service-oriented principles. Our empirical evaluation shows that fault-tolerance overhead is about 1% in terms of additional resources required thus making the approach feasible.

Keywords: Scalable, Fault-tolerance, Service Oriented Management, Architecture

1. Introduction

With the explosion of the internet, a new class of Web-based applications has emerged. These applications have connected end users to existing, traditional, centralized services. Distributed applications today are composed of multiple distributed components and are increasing in complexity. As the individual components get widely dispersed, they tend to span different administrative domains. Differing network and security policies restrict access to application components while resource management access is further limited due to presence of network firewalls and *Network Address Translation (NAT)* devices. Further, different services may be running on different platforms and could have been written in different languages. As application complexity grows, the need for an efficient management system emerges.

Various system specific management architectures have been developed and have been quite successful in their areas. Examples include SNMP (Simple Network Management Protocol) [1] CMIP [2] and CIM [3]. The Java community has introduced JMX [4] (*Java Management eXtensions*) which enables any Java-based resource to be automatically manageable. WMI [5] (*Windows Management Instrumentation*) from Microsoft enables local and remote monitoring and management of Microsoft Windows based machines. A main lacking feature among these management systems is *interoperability*.

In this paper we propose a simple, universal mechanism for managing a set of distributed entities. Every entity implicitly has or can be explicitly augmented with a Web service interface. The only assumption in providing fault-tolerance is the existence of a scalable and reliable database for storing system state. Our current implementation leverages the WS-Management [6] specification, but could very well use WS – Distributed Management (WSDM) [7]. WS

Management was primarily chosen for its simplicity and also to leverage an existing implementation of WS-Eventing [8] in the NaradaBrokering [9] project.

1.1. Service-oriented Management

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. Proposals [10] that leverage the Web Services management principles in context of existing management frameworks already exist. The service-oriented architecture provides a simple and flexible framework for building sophisticated applications. The use of XML in implementing Web Services facilitates interactions between services implemented in different languages, running on different platforms and communicating over multiple transports.

WS Management and WSDM 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 [11]) and Management of Web Services (MOWS [12]). MUWS focuses on providing a unifying layer on top of existing management specifications such as CIM, SNMP and OMI (Open Management Interface) 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 (Service Level Agreement) 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`** resources, **`GET`** and **`PUT`** individual settings and dynamic values, **`ENUMERATE`** contents of containers and collections, **`SUBSCRIBE`** to events emitted by managed resources and **`EXECUTE`** resource specific management actions. Thus the majority of overlapping areas with the WSDM specification are in the MUWS specification. Ref. [13] presents a proposal for evolution of a common management specification.

1.2. Generic Management

Application components require specific configuration to provide optimum Quality of Service (QoS). The *"Component-Specific Configuration"* is usually dependent on user-defined criteria and may require components to be individually configured. As again components present behind firewalls are usually unreachable via standard means.

Figure 1 shows the basic components of a generic management framework. We assume that the resource to manage and the resource manager are Web Services.

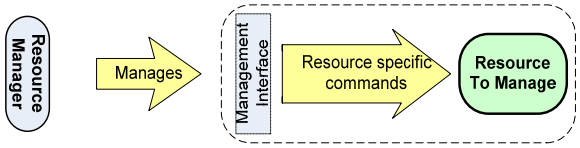

Figure 1 Generic Management Framework

The *Resource* that requires 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 commands.

Management of resources includes[1] resource configuration and performing life-cycle operations such as *CREATE* and *DELETE* resource instances whenever applicable. Management also includes processing runtime events, monitoring status and performance of the resources and maintaining system state as defined by some user-defined criteria. Management operations change system state which raises a number of consistency considerations not present in monitoring systems such as MonaLISA **[14]** and Astrolabe **[15]**.

In our architecture, we assume there could be multiple such services that require management. Examples of systems with large number of manageable resources are cell phones, large clusters of machines or even brokers in distributed brokering systems. The scheme should thus be scalable, incorporating management of a large number of distributed resources. Further, as systems span wide networks they become difficult to maintain and failure is norm. Systems must have the capability to detect failure and restart the failed service (resource) or re-instantiate a copy of the failed component that takes over the functionality of the failed resource.

The rest of the paper is organized as follows. We present our architecture in Section 2 and present evaluation results in Section 3. Section 4 describes the application of our architecture towards managing a grid messaging middleware. We present a summary of existing fault-tolerance strategies in section 5. In section 6 we present our conclusion and future work.

2. Architecture

Our architecture is based on existing fault-tolerance schemes. The approach uses intrinsically robust and scalable management services and relies only on the existence of a reliable, scalable database to store system state.

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. 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 components is shown in **Figure 2**. The function of various components is discussed below:

[1] From WS – Distributed Management, *http://devresource.hp.com/drc/slide_presentations/wsdm/index.jsp*

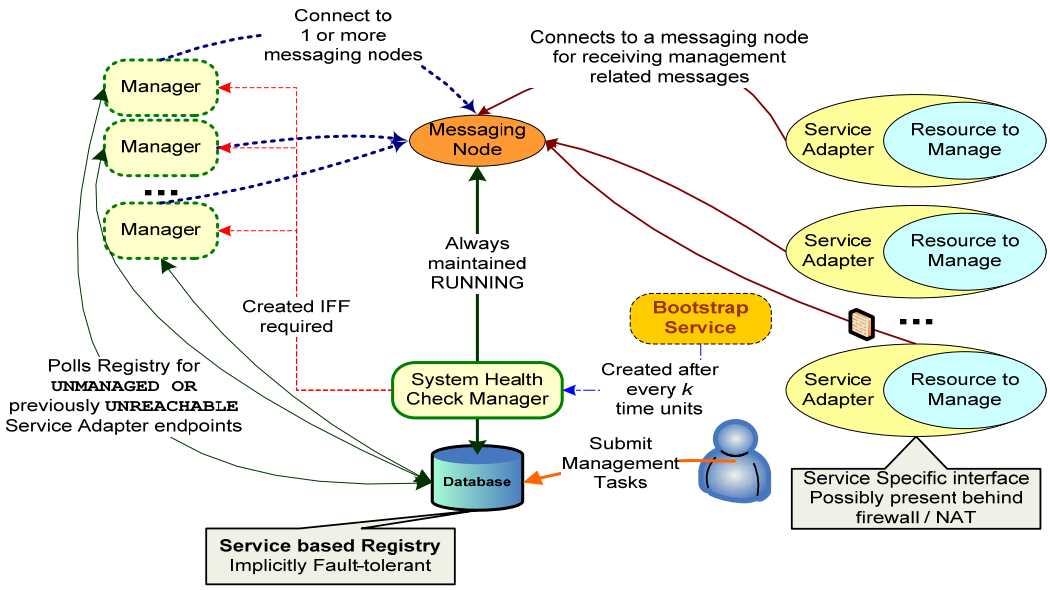

Figure 2 Overview showing components of the Management Architecture

2.1. Resource

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 port 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 resources of the Resource.

2.2. Service Adapter

Service adapter serves as a mediator between the manager and the Resource. We assume that there is one Service adapter per 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). 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 [16].
3. Hosting a service oriented messaging based management processor protocol such as WS Management (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.

2.3. 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. This resource specific manager independently commits runtime state of the resource to the registry. Thus, the manager process implements the *Independent Check-pointing* scheme. 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. Thus the architecture implements a *Passive Replication* scheme. The primary purpose of independent check-pointing and passive replication is simplicity of implementation.

2.4. 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 [17] service. Usually read operations can be directly served from an in-memory cache but writes are always written directly to the persistent store. The presence of a persistent store provides fault-tolerance to the registry service. We assume the persistent store to be distributed and replicated for performance and fault-tolerance purposes.

A *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, serving as a matching engine to match new resources to managers and updating appropriate fields in the metadata maintained by the Registry.

2.5. 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.

2.6. 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.

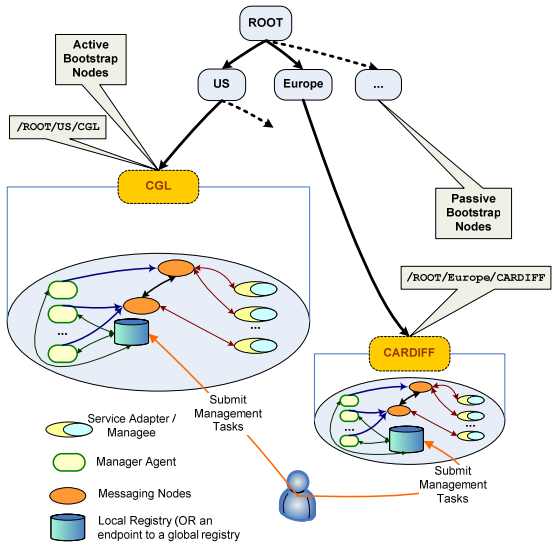

Figure 3 Achieving scalability through hierarchical management

The bootstrap services are arranged hierarchically as shown in **Figure 3**. 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). Such hierarchical arrangement is used to achieve scalability in many systems such as Domain Name Service (DNS) [18], Astrolable [15] and MonaLISA [14].

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. Since the number of child nodes per parent node is relatively small, the system may maintain a reliable connection (TCP) between the child and parent nodes. The child nodes may send periodic heartbeats to the parent node. Failure is quickly detected when there is a connection loss OR a heartbeat is not received within a specified timeframe.

2.7. 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.

3. Performance Evaluation

In this section we present analysis of the system and present benchmark results. Our system adds a few components apart from the actual resources being managed, in order to achieve scalable, fault-tolerant management. The main purpose of benchmarking analysis is to show the feasibility of the system. 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).

3.1. Maximum message rates

The measurements presented in this paper use a single NaradaBrokering messaging

node as a transport substrate. 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 **`4500 messages / sec`** when the message size is about 1024 bytes.

Since most of the message interactions comprise of messages which can be encoded using 512 bytes or less, 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.

3.2. 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.

3.3. Runtime Response Cost

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.

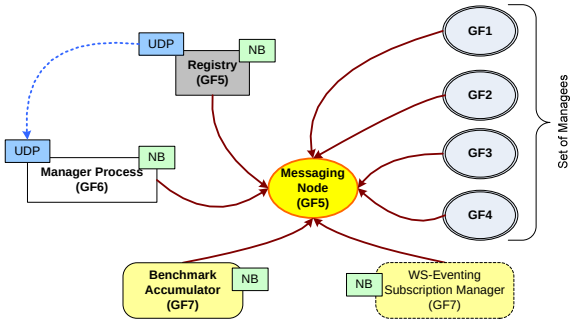

Figure 4 Test Setup

The test setup is shown in **Figure 4**. We ran multiple Resources from 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 and the subscription manager process for storing WS-Eventing subscriptions were run on GF7. As shown in the figure, only Manager – Registry interactions go over UDP. All other interactions are made by publishing / subscribing to the appropriate topic.

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 message). 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. Thus all the timings are higher than the actual timings by a few milliseconds atmost.

Figure 5 Response time of a single Manager process

As expected, with an increase in the number of managed resources, the number of threads per manager process grows. 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 average response time is shown in **Figure 5**. The figure shows the metrics when there is only 1 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 6**. However, 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.

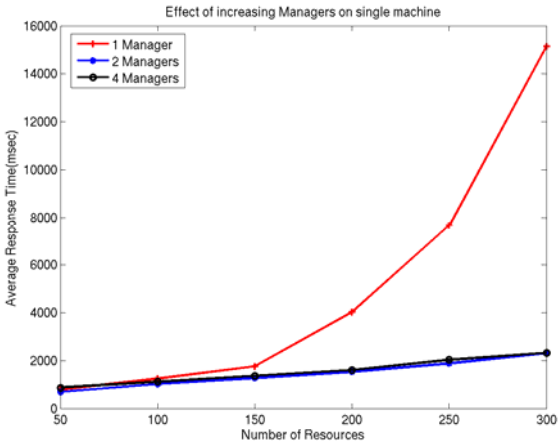

Figure 6 Increasing Manager processes on the same machine

Figure 7 Average response time improvement using 2 and 4 Manager processes

Figure 7 shows the performance (close-up of **Figure 6**) when there are 2 and 4 manager processes respectively. We conclude that adding more manager processes than the number of available processors on a particular node, does not necessarily improve system performance.

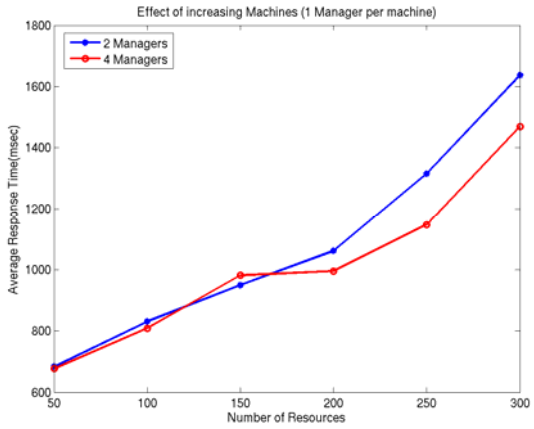

Figure 8 Effect of increasing number of machines

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 8** and **Figure 5**). As seen from **Figure 8**, the expected gain (increasing machines from 2 to 4) is not very high because of the use of a single messaging node through which all messages must pass, possibly leading to a bottleneck.

We note that as the number of managed resources increases beyond 200, the response time increases rapidly. Thus we conclude that a single manager can manage up to 200 resources with an acceptable delay.

3.4. 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 can be managed by a single manager process, then we require at-least **N/D** manager processes. Each manager process requires 1 connection to the messaging node. Also, the registry may maintain a connection to the messaging node. Thus the total number of connections is **N + N/D + 1**. If **Z** is the maximum number of connections a single messaging node can handle, then we have

N + N/D + 1 <= Z,
or N <= (Z-1)*D/(1+D).

In our measurements a single broker could reliably support about 800 simultaneous TCP connections. So assuming that a single manager process can manage up to 200 resources, the total number of resources that can be managed are approximately 795. To scale to a larger number of resources, either a different protocol such as UDP may be used or a hierarchical arrangement of brokers may be employed. The second approach however 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.

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 are 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 a direct connection to the resource (e.g. via HTTP/TCP). This provides the basis for leveraging a *"publish-subscribe"* based messaging substrate such as NaradaBrokering.

Let **Z** be the maximum number of resources that a single messaging node can support. Thus to manage **N** resources we require

CEILING (N/Z) messaging nodes. The value of **Z** depends on the type of transport used. Thus if TCP is used, a single messaging node can support up to **800** resource connections. Using UDP however this value jumps up to **1500** or more.

Thus we require **N/Z** leaf domains (Refer Section 2.6). Again, if **D** is the maximum number of resources that can be managed by a single manager process, then we need at least **Z/D** manager processes per leaf domain. Further a single leaf domain would also have its own boot strap node and possibly one registry (or registry endpoint).

Thus total number of management infrastructure processes in a single lowest leaf level is

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

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 + Z/D)*N/Z** processes. Hence total processes

= N Resource + Management Infrastucture
= N + (2 + Z/D)*N/Z

Thus, the percentage of management infrastructure required is

$$MGMT_{INFRASTRUCTURE}$$
= [(2+Z/D)*N/Z] / [N + (2+Z/D)*N/Z]
= [1 – 1/(1 + 2/Z + 1/D)] * 100 %

As an illustration, if **D = 200** and **Z = 800**, then **MGMTInfrastructure**

= [1–1/(1 + 2/800 + 1/200)] * 100 %
= [1 – (400/403)] * 100 %
= (3/403) * 100 %
≈ 0.75 %

Thus we can conclude that as the number of resources to manage increases, fault-tolerant management of the system can be achieved by adding **0.75%** more resources. This makes the approach feasible.

4. Application to a Grid Messaging Middleware

The system feasibility as determined in the previous section is mainly dependent on the run-time state maintained per resource-specific manager thread and the number of registry accesses required to retrieve / store state. We believe that the analysis presented is applicable to a wide variety of resources where management can be done by maintaining a small amount of run-time state and a very small number of in-frequent registry accesses. One such application is management of a Grid Messaging Middleware: NaradaBrokering.

A previous version of this paper [19] focused on the motivation for management and proposed a general service-oriented management framework based on WS-Management. This paper primarily focusses on fault-tolerant aspects of management. NaradaBrokering [20] 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. One crucial application is in the context of audio-video collaboration system where there could be thousands of participants.

As an illustration, consider the problem of deploying a brokering network for supporting 10000 clients in a collaborative [21] fashion. Ref. [22] 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. 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.

4.1. State managed per Broker Resource

Each broker stores one instance of a *NodeInfo* object that contains information regarding the broker node's configuration and is usually between 1 to 2 KBytes. One *LinkInfo* object is maintained per outgoing link and this contributes up to 512 Bytes. Typically the maximum number of links is dependent on the topology. For instance, in a ring topology there is only 1 outgoing link from each node while for a 3 level tree topology, the number of outgoing links range from 0 to 3. This assumes that at each level, a chain of nodes is maintained. Finally, for each *NodeInfo* and *LinkInfo* object, a *ResourceLog* is maintained that corresponds to the system state that needs to be written to the registry at regular intervals. Typically the size is maintained around 2 Kbytes. Thus assuming 3 links, the total state size maintained per broker resource is **(2048 + 3 * 512 + 4 * 2048 = 12 Kbytes)**. This state is small enough so that it can be read / written to registry in typically 1 call.

4.2. Deployment Costs

We timed the initialization cost per broker node. There are 2 distinct costs associated with the broker as shown in **Table 1**. In the cold-start mode, the JVM needs to read classes from associated class files and so all operations take much higher time. In the hot-start phase (when a broker network is torn down and brought back up with possibly different configuration / topology), the JVM is already in an initialized phase and so these modifications take significantly less time.

Operation	Cold-start Time (msec)	Time when initialized (msec)
Set Config	1110	46.75
Create Broker	734	132.75
Create Link	94	43.00
Delete Link	109	35.25
Delete Broker	110	187.50

Table 1 Time per operation

Number of Nodes	Total Links in the network	Overall Time (msec)
Ring:		
8	**8**	**12515**
4	**4**	**8906**
1	**(NO LINKS)**	**1156**
Cluster:		
8	**7**	**15968**

Table 2 Topology deployment times

The time required to deploy a topology is primarily dependent on the time required to establish links. This is because, links introduce a dependency and this is an artifact of NaradaBrokering. The dependency arises when a broker A tries to establish a link to broker B when broker B is not initialized (not ready to accept incoming connections). Broker A then waits for some time and retries the connection. We measure

the time it requires to establish a ring topology with 1, 4 and 8 nodes and a cluster topology with 8 nodes. The results are summarized in **Table 2**. We have 7 links for cluster topology since the setup had 3 nodes per cluster connected in a chain (3 clusters, 5 links), 2 clusters per super cluster (2 super-clusters, 1 link) and 2 super-clusters per super-super cluster (1 link).

Note that these values are typical and would differ in a different setup and time of experiment.

5. Fault Tolerance in Distributed Systems

Faults in distributed systems are normal and it is desired that the system continues operation in presence of failures. Fault-tolerance is defined **[23]** as the characteristic by which *"A Distributed System can mask the failure occurrence and recover from failure"*. Distributed systems have addressed fault-tolerance of application components via strategies such as replication and check-pointing. In this section we present an overview of these schemes.

5.1. Replication

Replication schemes provide seamless transfer of control to a new or exiting 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 requires extra 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* **[24]** or using a central coordinator that functions as a *Sequencer* can be used, they suffer from scalability problems. Ref. **[25]** presents a hybrid approach for achieving *Totally Ordered Multicast* in large scale systems.

5.2. 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* **[26]**, *XCAT* **[27]** and MPI based message passing system such as *LAM-MPI* **[28]** 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 **[29]** 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* **[30]** snapshot of the system's state. A survey of various roll-back and recovery protocols can be found in **[31]**. We summarize the main techniques 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 it 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. [32] and [33] provide details on implementation of blocking coordinated check-pointing. A non-blocking coordinated check-pointing algorithm that uses application level check-pointing is presented in [34].

5.3. Fault-tolerance in Object-based Distributed Systems

Object based Distributed systems are an extension of the object-oriented programming systems. As the name suggests a *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) [35] from Microsoft, Common Object Request Broker Architecture (CORBA) [36] from OMG (Object Management Group) and Java / Remote Method Invocation (Java/RMI) [37] from Javasoft.

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 replicates objects into object groups consisting 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. A *Replication Manager* is responsible for creating and managing a group of replicated objects using a variety of different replication strategies. This manager in turn can be replicated for fault-tolerance.

The object oriented nature of Java facilitates code reuse and significantly reduces development time. JVM however does not support fault-tolerance. Fault-tolerance is enabled by using systems such as Nomads [38] 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. [39] describes an approach to make check-pointing JVM independent by modifying the program's bytecode rather than modifying the JVM.

6. Conclusion and Future Work

In this paper we have presented a scalable, fault tolerant management framework. To make the management framework interoperable we employed a service-oriented architecture based on WS-Management. Our experimental evaluation shows that as the number of resources increases beyond 200, fault-tolerant management can be achieved using only 0.75% more processes. This feasibility is a

result of using NaradaBrokering itself as a *"publish-subscribe"* based 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.

Our current implementation and feasibility is based on the requirement of a small runtime state which can be read / written to a persistent registry in as few calls as possible. In the future we would like to apply the framework to areas where this assumption need not necessarily hold true (such as large scientific application). We believe that application of management framework to such systems can bring up many interesting research issues, specifically challenging scalability of the system.

7. References

1. 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.
2. 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.
3. Distributed Management Task Force, I. *Common Information Model*. Available from: http://www.dmtf.org/standards/cim/.
4. Kreger, H., *Java Management Extensions for application management.* IBM Systems Journal, 2001. **40**(1).
5. Microsoft. *Windows Management Instrumentation (WMI)*. Available from: http://www.microsoft.com/whdc/system/pnppwr/wmi/default.mspx.
6. 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.
7. HP. *Web Services Distributed Management (WSDM)*. March 2005, Available from: http://devresource.hp.com/drc/specifications/wsdm/index.jsp.
8. Microsoft, IBM, and BEA. *Web Services Eventing (WS – Eventing)*. Aug 2004, Available from: http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.

17. 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.

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

19. 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.

20. 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.

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

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

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

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

25. Rodrigues, L., H. Fonseca, and P. Verissimo. *Totally Ordered Multicast in Large-Scale Systems*. in *16th Intl. Conf. on Distributed Computing Systems*. 1996.

26. *Condor Project*. Available from: http://www.cs.wisc.edu/condor/.

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

28. *LAM-MPI*. Available from: Project Page: http://www.lam-mpi.org.

29. 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.

30. 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.

31. 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.

32. 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.

33. 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.

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

35. 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.

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

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

38. 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.

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