Scalable, Fault-tolerant Management of Grid Services

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. Use of services existing on different platforms, implemented in different languages and presence of variety of network constraints further complicates management.**

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 for providing scalable, fault-tolerant management framework.

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

I. Introduction

The 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.

A. 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 entities on the network. We consider a specific case of distributed applications where such entities can be controlled by zero or modest external state. We define modest state as being one which can be exchanged using very few messages (typically 1 message exchange). These entities in turn can bootstrap and control components with much higher state such as software services. We consider the combination of such an entity and the component associated with it as a *Manageable Resource*. An example of such an entity is a *Broker Service Adapter* which bootstraps a broker service in a distributed messaging system using about 16KB of external state [6]. Ref. [7], presents an application of the proposed framework in context of management of distributed messaging middleware.

Next, we discuss the scope of management with respect to the discussion presented in this paper. The primary goal of *Resource 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 resource utilization and other that deals with resource administration.

In the first category, resource management provides resource allocation and scheduling with a specific goal such as *sharing resources fairly while maintaining optimal resource utilization*. For example, operating systems [8] provide resource management by providing fair resource sharing via process and memory management. Condor [9] provides specialized job management for compute intensive tasks. Similarly, Grid Resource Allocation Manager [10] 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. Further to avoid possible conflict with other definitions of the term *"resource"* elsewhere in literature (e.g. WS-Resource as defined by WSRF), we use the term service to mean any manageable entity. The approach presented in this paper is applicable to services which explicitly or implicitly provides appropriate management interface.

B. Motivation

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 services. Management is required for maintaining a properly running application; however existing approaches have shown limitations to successfully manage such large scale systems.

First, the number of components susceptible to failure increases as the size of an application increases in terms of factors such as hardware components, software components and geographical scale. An analysis [11] of causes of failure in Internet services shows that most of the service's downtime can be attributed to improper management (such as wrong configuration) while software failures come second.

Second, with the growing size and complexity of applications, the cost of administration is increasing while the difficulty of administration tasks approaches the limits of an administrator's skills.

Third, different types of services in a system require different service specific management frameworks. As previously discussed, the management systems have evolved independently. This complicates application implementation by requiring the use of different proprietary technologies for managing different types of services 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 service with modest external state. We implement Web service based protocols to provide interoperable management. This enables us to effectively integrate existing management systems. We employ a hierarchical bootstrapping mechanism to scale the management framework over a wide-area. The framework is tolerant to failures within the framework itself while service failure is handled by executing user-defined failure handling policies. Finally, we evaluate our system to show that to manage N services we require about 1% additional resources which correspond to the components of the management framework.

C. 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. Service failure is the norm. Failure could be a result of the actual service failure or because of failure of some related component such as network making service inaccessible or even because of the failure of the management framework itself. While the framework must provide *self-healing* capabilities, service failure must be handled by providing appropriate policies to detect and handle failures while avoiding inconsistencies.

Scalability: The framework must scale as the number of manageable services increases. Also, the management framework itself adds additional components which are required to provide features such as fault-tolerance. The framework must also scale in terms of additional management framework components.

Performance: Runtime events are generated by services and the management system takes a finite amount of time to respond to faults. The challenge is to achieve acceptable performance in terms of recovery from failure and responsiveness to faults as the number of manageable services and the additional components required increases.

Interoperability: As previously discussed, services exist on different platforms and may be written in different languages. While proprietary management systems such as JMX and Windows Management Instrumentation [12] have been quite successful, they are not interoperable limiting their use in heterogeneous systems and platforms. The management framework must address the interoperability issue.

Generality: Service management must be generic, i.e. the framework must apply equally well to services that wrap hardware as well as software services.

Usability: The management framework must be usable in terms of autonomous operation provided by the framework whenever possible. The framework must automatically detect and correct failures within the management framework itself by automatically 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 II. We describe our test setup and evaluation of the framework in Section III. We also discuss the feasibility of the system. We present related work in Section IV. Section V is conclusion and future work.

II. ARCHITECTURE

Our approach uses intrinsically robust and scalable management services and relies only on the existence of a reliable, scalable service to store system state. The system leverages well known strategies for providing fault-tolerance (periodic check-pointing of system state, passive replication and request retry), scalability (hierarchical organization, asynchronous communication) and failure detection (service heartbeats).

A. Hierarchical Distribution

The overall management framework consists of units arranged hierarchically (refer Figure 1). Each unit is controlled via a statically configured service called the bootstrap node. These nodes are assumed to be internally replicated to provide fault tolerance. The hierarchical organization of units makes the system scalable in a wide-area deployment. Scalability is achieved by having each individual bootstrap nodes manage only the services for which it is responsible for. The hierarchical organization is shown to scale in existing systems such as the Domain Naming System [13].The tree hierarchy only exists to ensure that all leaf bootstrap nodes are always up and running and to take appropriate action to mitigate failure. The bootstrap node mainly exists to serve as a starting point for all components of the system.

The bootstrap node also functions as a key fault-prevention component that ensures the management architecture is always up and running. This is done by periodically spawning a health-check service that checks the system health and if some component has failed, the health-check service reinstates that component. The leaf nodes of the hierarchy are active bootstrap nodes. These nodes are responsible for maintaining a working management framework for the specified set of machines (domains). 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.

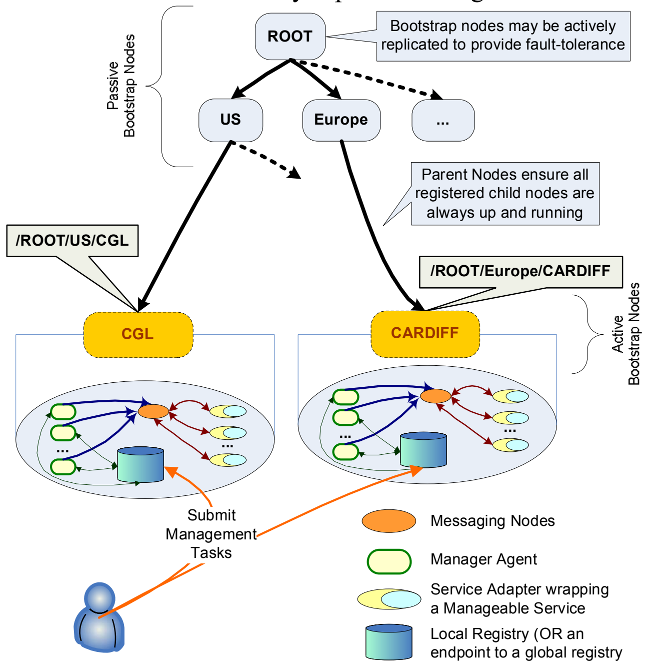
Passive Bootstrap Nodes
ROOT
Bootstrap nodes may be actively replicated to provide fault-tolerance
US
Europe
...
Parent Nodes ensure all registered child nodes are always up and running
/ROOT/US/CGL
/ROOT/Europe/CARDIFF
CGL
CARDIFF
Active Bootstrap Nodes
Submit Management Tasks
Messaging Nodes
Manager Agent
Service Adapter wrapping a Manageable Service
Local Registry (OR an endpoint to a global registry

Figure 1 Achieving scalability through hierarchical arrangement

Note that the individual units are functionally identical. Thus while they work on separate sets of services and have separate manager processes, they are expected to perform similarly in situations such as responding to runtime events.

B. Components of a unit of management framework

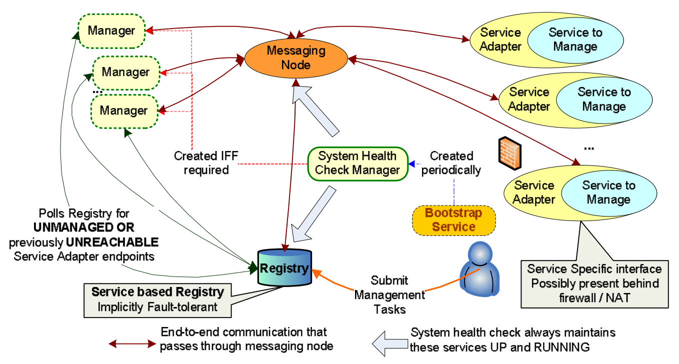
Manager
Manager
Manager
Messaging Node
Service Adapter
Service to Manage
Service Adapter
Service to Manage
...
Service Adapter
Service to Manage
Created IFF required
System Health Check Manager
Created periodically
Bootstrap Service
Polls Registry for UNMANAGED OR previously UNREACHABLE Service Adapter endpoints
Registry
Service based Registry Implicitly Fault-tolerant
Submit Management Tasks
Service Specific interface Possibly present behind firewall / NAT
End-to-end communication that passes through messaging node
System health check always maintains these services UP and RUNNING

Figure 2 Overview showing components of a unit of Management Framework

We now describe the main components of each domain of the framework. The unit of management framework consists of one or more manageable services, their associated service managers, one or more messaging nodes (to provide a scalable messaging substrate) and a scalable, fault-tolerant database which serves as a registry.

The arrangement of these components is shown in Figure 2. We now describe each of these components in detail. We will then provide an overview of the consistency scheme and means to address security issues in the system.

1) Service: As defined in Section I.A we refer to service as the component that requires management. We employ a service-oriented management architecture and hence we assume that these services implicitly or explicitly have an appropriate management interface.

2) Service Adapter: Service adapter serves as a mediator between the service manager and the service. 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 [14]
3. Hosting a service oriented messaging based management processor protocol such as WS Management (refer [15] for details on our implementation). The WS – Management processor provides basic management framework while the wrapper provides the correct functionality (mapping WS Management messages to service-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 implementing user-defined policies to address failure. Such policies are executed by a service-specific manager.

3) Manager: A manager is a multi threaded process and can manage multiple services at once. Typically, one service-specific manager module thread is responsible for managing exactly one service and is also responsible for maintaining the service's configuration. Since every service-specific manager only deals with the state specific to the service it is managing and the runtime state is modest, it can independently save this state periodically to the registry. Manager processes usually maintain very little or no state as the state can be retrieved easily by either querying the service 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 services and if a manager process has not renewed its existence within a specified time, all services 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 service, the manager always verifies if it is still responsible for managing the service before re-establishing management ownership with the service. This helps in preventing two managers from managing the same service.

4) Registry: The registry stores system state. System state comprises of runtime information such as availability of managers, list of services 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 is assumed to be scalable and fault-tolerant. The registry also 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 service and manager and assigning services to managers.

5) Messaging Node: Messaging nodes consist of statically configured NaradaBrokering [16] broker nodes. The messaging node provides a transport-independent scalable message routing substrate to route messages between the managers and service adapters. Scalability comes from the use of a publish/subscribe based communication system that multiplexes requests to multiple endpoints using a single connection channel. NaradaBrokering can use multiple types of transports such as TCP, UDP, HTTP and SSL. This allows a service, 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 fall back to the backup messaging node. Further, these nodes rarely require a change of configuration and can be restarted automatically using the default static configuration for that node.

6) User: The user component of the system is the service requestor. A user (system administrator for the services being managed) specifies the system configuration per service. This information is consistently maintained by the registry. The managers are then responsible for bootstrapping and managing the services appropriately in accordance with the user-defined configuration. The user also defines policies to handle service-specific events including failures. These are executed by the associated service-specific manager.

C. 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 service
2. Old messages reaching after new requests
3. Multiple copies of same service existing at the same time. This especially happens when a user-defined policy dictates automatic instantiation of a service after it is deemed unreachable for more than the specified time duration.

To handle these issues, we assume the request processor in registry to generate a monotonically increasing unique *Instance ID* (IID) for each instance of service (managed service or service 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. An entity stores the last successfully processed message's message id (which is composed of the sender's IID and a monotonically increasing sequence number) allowing it to distinguish between duplicates and obsoletes.
3. A service adapter periodically renews its existence within the registry. In response to the renewal, the registry sends back the currently known instance id of the service. When multiple copies of same service exist, the older service instance's service adapter sees that its instance id is < current instance id and shuts down the service.

D. Security

A distributed system gives rise to several security issues such as but not limited to denial of service, unauthorized access to services 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 [17] ensures that physical location (host and port) of any entity is never revealed. The security framework [18] 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.

III. Performance Evaluation

In this section we analyze the system from scalability point of view. Failure recovery of framework components is currently based on timeouts while correctness is handled as described in Section II.C. 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). We implemented the WS-Management specification to facilitate communication between the service manager and the service' service adapter.

The most important deciding factor which determines the maximum number of requests a manager process can handle is how the response time varies as the number of services being managed by a single process increases. In this section, we evaluate our system by observing the worst case scenario (where all services managed by a manager process generate events concurrently) for a single unit of framework. We expect the individual units (leaf domains) in the framework to perform similarly given similar set of conditions (machines, network and service behavior and requirements).

Further, this would also enable us to formulate the number of manager processes that are required and the number of services that can be managed within a single domain (unit) of the management architecture.

A. Test Setup and Observations

The test setup is shown in Figure 3. As shown in the figure, we increase managers on a single machine (Setup A) and multiple machines (Setup B). The testing methodology was as follows. The services run via a thread-pool that sends pre-generated events to the managers. A timer is started just before sending these events. In response to the events, the service-specific manager thread responds back to the service. When all services get their corresponding response, we stop the timer and the difference corresponds to the overall response time.

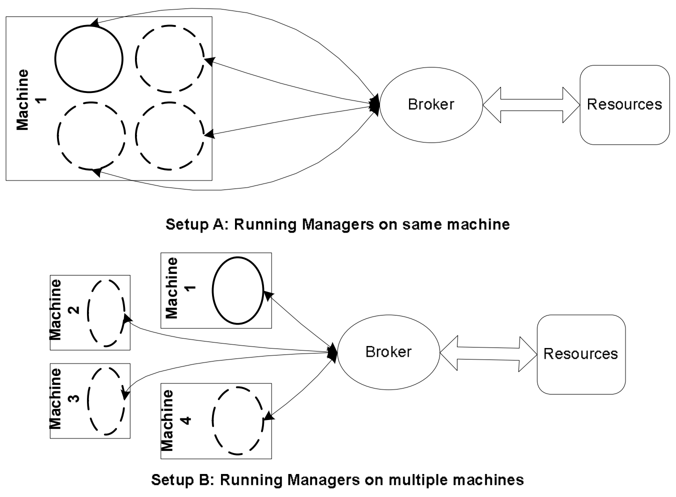
Machine 1
Broker
Resources
Setup A: Running Managers on same machine
Machine 1
Machine 2
Machine 3
Machine 4
Broker
Resources
Setup B: Running Managers on multiple machines

Figure 3 Test Setup

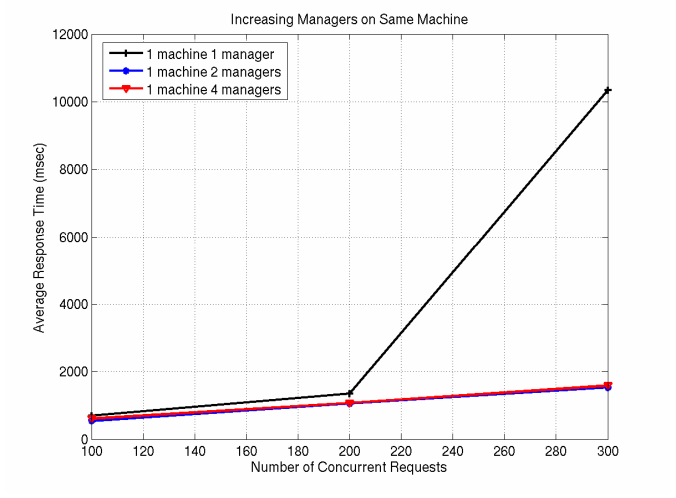

Figure 4 Average response time when handling multiple concurrent requests by increasing manager processes (on the same machine)

The measured response time shows a case with catastrophic failures, one in which every single service being managed generates an event. As expected, with an increase in the number of managed services, the average response time increases. In our case, there was no registry access during processing of the event, however this behavior is service specific and may require one or more registry accesses in certain cases.

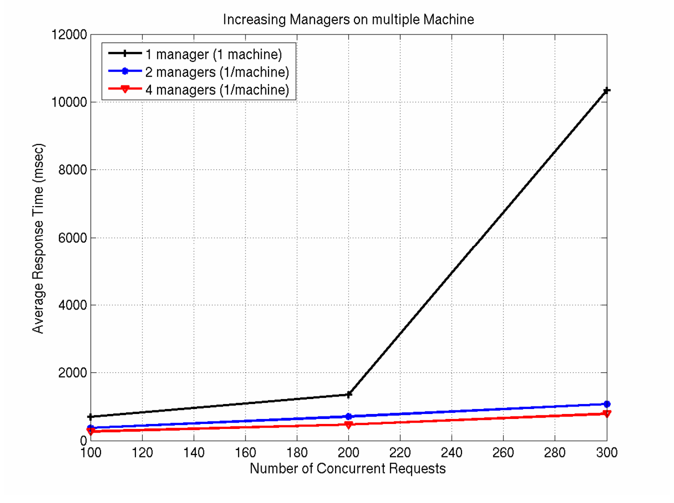

Figure 5 Average response time when handling multiple concurrent requests by increasing manager processes (1 per machine)

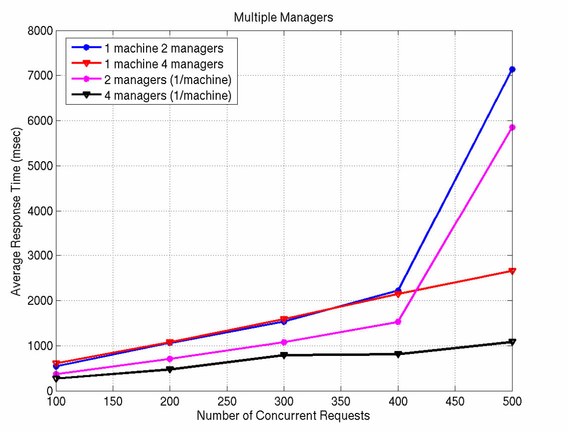

Figure 6 Response time comparison (Increasing manager processes on same vs. multiple machines)

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. As we increase the number of managers, we see a huge performance benefit by increasing the processes from 1 to 2 as the number of concurrent requests increases beyond 200. We note that, if 4 manager processes are used on same machine instead of 2, we see that the average response time slightly. 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 concurrent events.

In case of increasing managers on multiple machines, one manager per machine (refer Figure 5), there is a performance improvement when using 4 managers instead of 2. Figure 6 shows the performance of 2 and 4 managers when we increase the number of concurrent requests beyond 300. We note that a similar pattern emerges whereby a single manager saturates when the number of concurrent requests increases beyond 200.

B. Performance Model

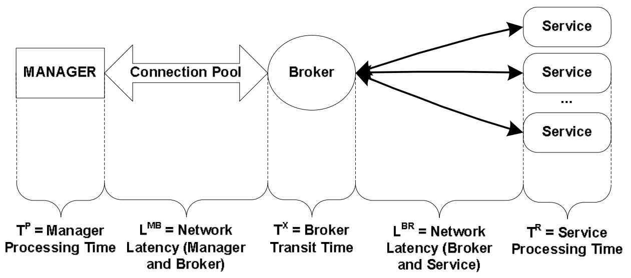
MANAGER
Connection Pool
Broker
Service
Service
...
Service
T^P = Manager Processing Time
L^{MB} = Network Latency (Manager and Broker)
T^X = Broker Transit Time
L^{BR} = Network Latency (Broker and Service)
T^R = Service Processing Time

Figure 7 Model representing components of average response time as seen by services

Figure 7 shows the components of the average response time. We now discuss the various factors:

To compute the broker transit time (T^X), we measured the maximum throughput a single broker process can provide. We note that a single broker when not saturated, can give a throughput up to 5000 messages / sec for payload size of 512 bytes and in excess of 4500 messages / sec for a payload size of 1024 bytes. Thus on average T^X is < 1 ms and can be considered a constant if the broker is not saturated. The latencies between manager and broker (L^{MB}) and between broker and service (L^{BR}) is dependent on the network conditions but can be considered an unvarying constant for an unsaturated network. Since we use pre-created requests, the service processing time T^R corresponds to un-marshalling of the received response only and can be considered a constant. Thus we get

$$\begin{aligned} \text{Time} &= T^P + [T^R + 2*(L^{BR} + L^{MB} + T^X)] \\ &= T^P + K \end{aligned}$$

where the 2 multiplier corresponds to a request and response.

Further T^P (which is the processing time at the manager) is composed of the following 3 components:

T^{CPU} = a service-specific activity that includes necessary processing of the event including un-marshalling of the event request and marshalling of the corresponding response

$T^{EXTERNAL}$ = time taken when additional services such as registry requests are invoked in order to process the event.

$T^{SCHEDULING}$ = time taken by a processor to schedule the manager processes. This factor becomes significant when there are a lot of CPU intensive processes (e.g. more manager processes than the number of available processors). Thus in our test setup we note that the average response time for 4 managers is slightly higher than the average response time for 2 manager when running on a single machine (refer Figure 4). Thus,

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

In our case, we did not have any dependency on external service for processing an event and hence $T^{EXTERNAL} = 0$. Further, for a single manger process, $T^{SCHEDULING}$ is very small and can be ignored.

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

Also, the maximum request processing rate by a single multithreaded manager process is

$$D = C/T^P \text{ requests / sec.}$$

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

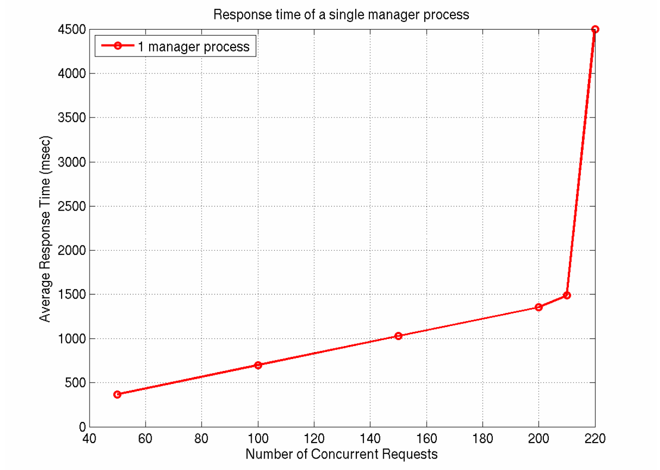

Figure 8 Saturation point for a single manager

As an illustration, we collected the average event processing times for 150 services using a single manager on 1 machine and we get the average value of T^P to be ≈ 8.37 msec. Our test setup ran on hyper-threaded CPUs which can run 2 threads simultaneously. Thus, we get

$$\begin{aligned} D &= C/T^P * 1000 \\ &= (2/8.37) * 1000 \approx 239 \text{ req. / sec.} \end{aligned}$$

To determine the saturation point for the manager in our test setup, we steadily increase the number of concurrent requests. As shown in Figure 8, a single manager saturates around 210 concurrent requests.

C. Discussion

A single manager process can manage a large number of services however can only process a finite number of concurrent requests (run-time events) from managed services. To provide quality of service to managed services, we choose to put a limit on the number of services assigned to a single manager process. This limit is D (= 210) in our case, note that the number 210 is illustrative with respect to our test setup and could easily be higher especially in the case where event processing is involves interaction with external services. Finally as an illustration we note that, in order to prevent manager process saturation in the worst case scenario, one would consider assigning only about 210 services to a single manager process.

Further, most services 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 service being managed generates an event. Additionally, note that the benchmark results presented here deal with only one unit of management infrastructure. We expect multiple independent units in a wide area deployment (refer Figure 1) to behave similarly.

D. Amount of Management Infrastructure Required

We now try to answer the research question, *"How much Management Infrastructure is required to handle N Services?"* We define the term *"Management Infrastructure"* as the additional services required for providing fault-tolerant management.

Let **N** be the number of services requiring management. If **D** is the maximum number of services assigned to a single manager process, then we require at-least **N/D** manager processes. Let **M** be the maximum number of services that a single messaging node can support. Thus to manage **N** services 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 **M/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 services, 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 services such that they are managed in different management domains.

To manage the **N/M** leaf domains, an additional number of passive bootstrap nodes are required. Typically the number of passive nodes would be **<< N/M** and we ignore it for the purpose of this analysis. Thus for managing **N** services we require an additional **(2 + R + M/D)*N/M** processes. Thus, the percentage of management infrastructure required with respect to number of services **N** is

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

As an illustration, if **D = 200, R = 4** and **M = 800**, then

$\mathbf{MGMT}_{\mathrm{INFRASTRUCTURE}}$
$= [(2+4)/800 + 1/200] * 100\ \%$
$= 1.2\ \%$

Note that, when the number of services **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 services. Finally, as previously discussed, the value of **D** may be suitably adjusted which would determine the number of manager processes required which in turn affects percentage of extra services ($\mathbf{MGMT}_{\mathrm{INFRASTRUCTURE}}$).

IV. Related Work

The Web Services Resource Framework (WSRF) [19] 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) [20] and Management using Web Services(MUWS) [21]. By contrast, in our framework any service that needs configuration, lifecycle and runtime management and satisfies the condition of modest external state can be wrapped with a service interface to expose management capabilities. Management is provided by a complementary specification, WS-Management [22] which is a SOAP-based protocol for managing systems (including Web Services) and functionally overlaps with MUWS.

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 I.A, 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 noninteroperable. 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.

V. 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. 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 service failure is handled by implementing user-defined policies. When applied to services with modest external state, the approach is feasible since it adds about 1% additional services to provide fault-tolerant management to a large set of distributed services.

Management is enabled by leveraging WS Management which uses simple operations such as GET and PUT. These operations can be modeled in a REST-like [27] fashion, thereby, making the approach applicable to managing services in any service-based architecture including Web 2.0.

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. Our current implementation uses WS – Management. In the future we would like to investigate implementing the merged [28] 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 services.

Acknowledgement

This research is supported by grants from the National Science Foundation's Division of Earth Sciences project number EAR-0446610, and the National Science Foundation's Information and Intelligent Systems Division project number IIS-0536947.

References

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

[2] Case, J., et al. *A Simple Network Management Protocol (SNMP)*. 1990; Available from: 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)*. Available from: http://www.dmtf.org/standards/cim/.

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

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

[7] Gadgil, H., *Scalable, Fault-tolerant management of Grid Services: Application to Messaging Middleware*, in *Ph.D. Thesis, Computer Science*. 2007, Indiana University: Bloomington.

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

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

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

[11] 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.

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

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

[14] 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.

[15] Pallickara, S., et al., *A Retrospective on the Development of Web Service Specifications*, in *Securing Web Services: Practical Usage of Standards and Specifications*, P. Panos, Editor. 2006, Idea Group Inc.: University of Newcastle Upon Tyne.

[16] 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.

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

[18] 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.

[19] Czajkowski, K., et al. *The WS-Resource Framework*. May 2004.

[20] 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.

[21] 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.

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

[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; Available from: http://jcp.org/en/jsr/detail?id=262.

[27] Fielding, R.T. and R.N. Taylor. *Principled Design of the Modern Web Architecture*. in *22nd International Conference on Software Engineering (ICSE '00)*. 2000. Limerick, London: ACM Press.

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