J Supercomput (2007) 42: 83–106
DOI 10.1007/s11227-006-0034-z

Tycho: a wide-area messaging framework with an integrated virtual registry

Mark A. Baker · Matthew Grove

Published online: 9 March 2007

Abstract In a distributed environment remote entities, usually the producers or consumers of services, need a means to publish their existence so that clients, needing their services, can search and find the appropriate ones that they can then interact with directly. The publication of information is via a registry service, and the interaction is via a high-level messaging service. Typically, separate libraries provide these two services. Tycho is an implementation of a wide-area asynchronous messaging framework with an integrated distributed registry. This will free developers from the need to assemble their applications from a range of potentially diverse middleware offerings, which should simplify and speed application development and more importantly allow developers to concentrate on their own domain of expertise. In the first part of the paper we outline our motivation for producing Tycho and then review a number of registry and messaging systems popular with the Grid community. In the second part of the paper we describe the architecture and implementation of Tycho. In the third part of the paper we present and discuss various performance tests that were undertaken to compare Tycho with alternative similar systems. Finally, we summarise and conclude the paper and outline future work.

Keywords Asynchronous messaging · Virtual registry · Distributed application · R-GMA · Globus MDS4 · NaradaBrokering · Performance evaluation

1 Introduction

Tycho is an implementation of a wide-area asynchronous messaging framework that includes an integrated distributed registry. This combination allows Tycho to provide

M.A. Baker (✉)
School of Systems Engineering, The University of Reading, Reading, RG6 6AY, UK
e-mail: Mark.Baker@Computer.Org

M. Grove
DSG, The University of Portsmouth, Portsmouth, UK

a range of key services for wide-area distributed applications that can essentially publish and discover endpoints, as well as exchange information without the need for the developer to use multiple libraries. Existing solutions typically use one of many communication mechanisms, for example SOAP [1] or GridFTP [2] coupled with a separate registry such as UDDI [3] or LDAP [4] to provide service discovery. Tycho frees developers from the need to assemble their applications from a range of potentially diverse middleware offerings, which will simplify and speed application development and more importantly allow developers to concentrate on own their domain of expertise.

1.1 Original motivation

The resource-monitoring framework, known as GridRM [5], has a distributed architecture where information needs to flow between remote gateways. Rather than reinvent a means of discovering and asynchronously transferring data between these end-points, a mature package was sought. Two classes of software would typically be used to solve this type of problem, a registry system, such as LDAP, and a messaging system such as SOAP via Apache Axis.

A solution for the messaging part of the solution would be via some type Message-Oriented Middleware (MOM) system. MOM typically operates in a layer between the application and transport layers at both ends of a communication path. MOM is well suited for applications requiring asynchronous messaging services, and for service-based systems in general since it provides an abstract model for communication between services. MOM-based systems, however, typically do not include a registry so that producers and consumers of the services can publish, search and bind with each other; this is normally undertaken by a third-party registry, or it is assumed that multi-cast is available, or is hardwired. To provide a full solution to our problem there is also a need for a registry service that is flexible and could be adapted to provide a generic system.

Even though the original motivation behind this project was to find and integrate an exiting solution into GridRM, it became evident that most generic distributed applications have similar requirements. This has led us believe that there is a general need for a system that provides a scalable registry and integrated wide-area messaging support.

In addition, as the Open Grid Services Architecture (OGSA) gains increasing acceptance in the e-Science community, a system that combines MOM with a generic registry will be a key aspect of these Service-Oriented Architectures (SOA). For example, Pallickara et al. [6] have seen the usefulness of this approach, but have implemented a fairly restricted registry. This type of combined system has so far not received much attention. The main motivation for the research detailed in this paper is to demonstrate how our framework, known as Tycho, can provide a combined wide-area asynchronous messaging and registry system that can be incorporated into a SOA for wide-area distributed applications.

2 Related work

In this section the most appropriate and popular systems used by the Grid community to provide a registry and high-level messaging services are discussed. The first two systems (MDS4 and R-GMA) have the capability to provide the registry services, whereas the final system (NaradaBrokering) could potentially provide the messaging service.

2.1 MDS4

The Globus Toolkit's Monitoring and Discovery Service (MDS version 4) [7] is a Web Services Resource Framework (WSRF) based implementation of a wide-area information and registry service. MDS4 provides a framework that can be used to collect, index and expose data about the state of grid resources and services. Typical uses for MDS4 include making resource data available for decision making in job submission services or notifying an administrator when storage space is running low on a cluster. MDS4 is open source and is distributed under the Apache 2 license, which is an OSI compliant license.

MDS4 is made up of several components that replace MDS2, which was based on LDAP. An aggregator framework, which currently consists of two services, provides MDS4 functionality. The Index Service is used to collect and discover information about resources and a Trigger Service, which can be configured to perform actions based on resource information gathered. MDS4 provides a WSRF interface for clients to query and subscribe to the information collected.

Figure 1 shows a client's interaction with the MDS4. MDS4 uses a fixed schema to ensure compatibility between its components; every participating MDS4 service must be configured to use the same schema. The index information itself is stored in memory. The Grid services are deployed in a Tomcat container using the Globus Toolkit's functionality, they can interact with clients written in Java, Python or C. MDS4 is not used for messaging between system components, the Globus Toolkit can be used to deploy grid services which use WSRF for communications, which are based on Web Services using SOAP over HTTP.

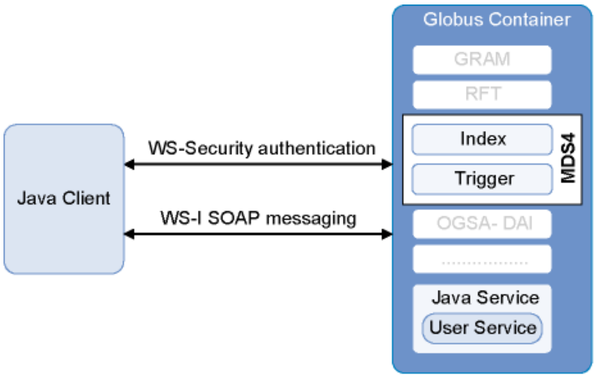


Fig. 1 The architecture of the globus toolkit with the two MDS4 services highlighted

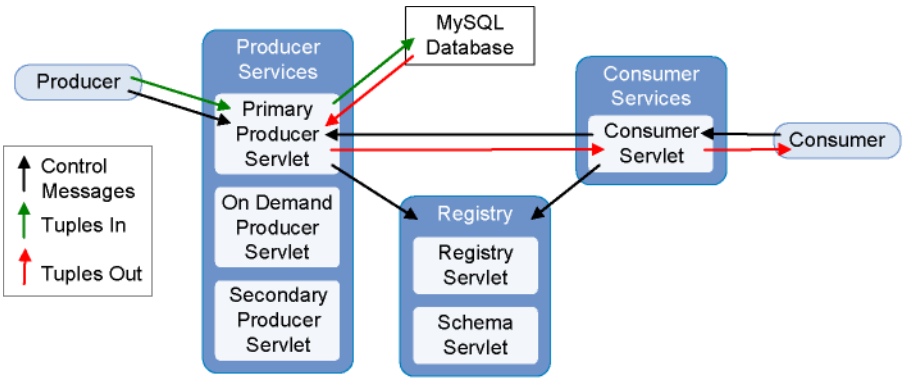


Fig. 2 The architecture of R-GMA

2.2 R-GMA (relational grid monitoring architecture)

R-GMA [8] is a Java-based implementation of the Grid Monitoring Architecture (GMA) [9] that was originally developed within the European DataGrid Project [10] for publishing network monitoring information over the wide-area and as an information service. R-GMA uses a relational model to search, using an SQL-like API, and describe the monitoring information it collects. It is based on a consumer/producer paradigm with client data being stored in a directory service, which presents it as a virtual database. R-GMA is now part of the Information and Monitoring work package of the EGEE project [11]. The current stable release of R-GMA (in gLite 1.5) is based on Java servlets; a more advanced version is under development using Web Services and SOAP for messaging. R-GMA uses the term tuples for sets of data being published or consumed. R-GMA can be used in conjunction with C++, C, Python and Perl consumers and/or producers, as well as obviously with Java.

Figure 2 shows an example of a consumer performing a continuous query for tuples published by a primary producer. The producer periodically publishes tuples to the primary producer service, which stores them for a period specified by the producer, this makes the tuples available for consumers to discover and consume. The consumer sends a query to the consumer service, which uses the registry to create a list of producers that can satisfy the query. The consumer service then sends a control message to the producer service requesting the tuples. As the tuples are delivered to the consumer service they are stored in a buffer for the consumer to collect. In the example shown in Fig. 2, the producer service is using a MySQL database to store published tuples and the consumer service is using in-memory storage to buffer the tuples for its consumer. Messaging between R-GMA components is provide using a customised Java NIO interface; this will be changed to SOAP over HTTP in a future release. R-GMA is distributed in a binary form, but it is possible to access the source code under the EGEE license, which is derived from a modified BSD license.

2.3 NaradaBrokering

The NaradaBrokering framework is a distributed messaging infrastructure, developed by the Community Grids Lab at Indiana University [12]. NaradaBrokering is an asynchronous messaging infrastructure with a publish and subscribe based architecture. NaradaBrokering is Sun JMS compliant. This messaging standard allows application components to exchange unified messages in an asynchronous system. The JMS specification is used to develop Message Orientated Middleware (MOM) and defines how messages are to be communicated via queues or topics. Networks of collaborating brokers are arranged in a cluster topology, with a hierarchy of clusters, super-clusters, and super-super-clusters. It aims to provide a unified messaging environment that incorporates the capability to support Grid and Web Services, Peer-to-Peer and video conferencing, within a SOA.

In NaradaBrokering, producers publish events belonging to a *topic* or *subject*, and then consumers can subscribe to those of interest to them. Topics in NaradaBrokering can be based on tag-value pairs, Integer and String values. In its simplest form these topics are typically "/" separated Strings. When a publisher issues events on a specific topic the middleware substrate routes the events to the subscribers that have registered an interest in this topic. Clients can specify SQL queries on properties contained in a JMS message. NaradaBrokering has a XML matching engine, which allows clients to specify subscriptions in XPath queries and store advertisements in XML encapsulated events. NaradaBrokering also provides a variety of transport protocols including HTTP, TCP, NIO/TCP, UDP, and SSL.

Figure 3 shows how brokers can be arranged hierarchically to provide a scalable messaging infrastructure. Each group of brokers is called a cluster, a cluster is a collection of connected brokers, a super-cluster is a collection of clusters and a super-super-cluster is a collection of super-clusters. The broker that is used for inter-cluster routing is called a controller. The hierarchical cluster architecture is designed to in-

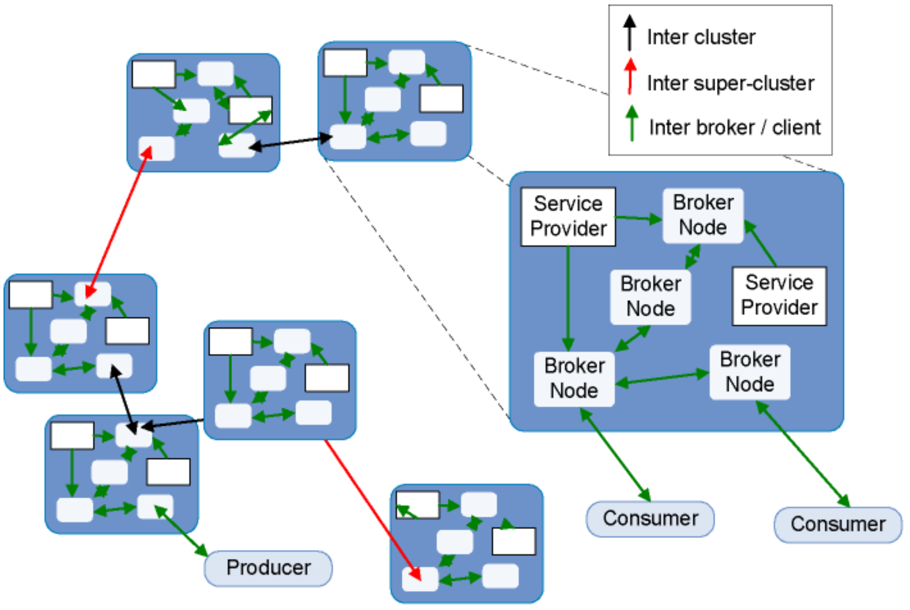


Fig. 3 The NaradaBrokering architecture showing communication between brokers, clusters of brokers and super-clusters

crease the scalability of the system; each broker is identified by a unique logical address, which is used to calculate a routing path through the network of brokers. Multiple routes can connect clients, brokers and clusters of brokers; each broker holds a network map, which is used to calculate the cost of routing a message to its destination. NaradaBrokering does not have a built-in general-purpose registry, but the latest release now has a topic and broker discovery service. NaradaBrokering is released under the Indiana University Advanced Research and Technology Institute License.

2.4 Summary

There are obviously a large number of systems that can provide a registry system or wide-area messaging services. In this section we have briefly described the architecture and functionality of the most popular systems used by the Grid community today. Our investigation has shown that no one system currently can provide both the registry and messaging functionality that we desire. In addition, there are a number of features and issues with each of these systems that make them less then desirable for our purposes (see the list below)—during the lifetime of the Tycho project these features and issues have obviously changed. However, the motivation for developing Tycho is still valid today, and can be summarised as:

- No one system provides both a scalable registry and messaging services—we believe this is a key factor that is novel and will ease application development.
- Each system discussed in this section is large and monolithic requiring expertise to install, configure and use.
- Architecturally the registry systems will not scale to Internet proportions and be able to cope with large number of resources and clients, without significant effort and additional programming.

Since our investigation failed to find one solution that satisfies our needs we have developed Tycho, which is an implementation of a wide-area asynchronous messaging framework with an integrated distributed registry. In the next section we described Tycho's architecture.

3 Tycho's architecture

Tycho is a Java-based framework based on a publish, subscribe and bind paradigm. We believe that a registry/messaging system should have an architecture similar to the Internet, where every node provides reliable core services, and the complexity is kept, as far as possible, to the edges. This implies that the core services can be kept to the minimum needed, and endpoints can provide higher-level and more sophisticated services, that may fail, but will not cause the overall system to crash. The design philosophy for Tycho has been to keep its core relatively small, simple and efficient, so that it has a minimal memory foot-print, is easy to install, and is capable of providing robust and reliable services. More sophisticated services can then be built on this core and are provided via libraries and tools to applications.

This will enable Tycho to be flexible and extensible so that it will be possible to incorporate additional features and functionality. Tycho's functionality has all been incorporated within a single Java JAR with the only requirement being a Java 1.5 JDK for building and running Tycho-based applications.

Tycho consists of the following components:

- Mediators that allow producers and consumers to discover each other and establish remote communications,
- Consumers that typically subscribe to receive information or events from producers,
- Producers that gather and publish information for consumers.

In Tycho, producers and/or consumers (clients) can publish their existence in a directory service known as the Virtual Registry (VR). A client uses the VR to locate other clients, which act as a source or sink for the data they are interested in. The VR is a distributed service provided by a network of mediators. When possible, clients communicate directly, however, for clients that do not have direct access to the Internet, the mediator provides wide-area connectivity by acting as a gateway or proxy into a localised Tycho installation. Figure 4 shows Tycho clients communicating between two remote sites connected via the Internet.

The Tycho VR is made up of a collection of services that provides the management of client information and facilitates locating and querying remote Tycho installations. A client registers with a local mediator, part of the VR, when it starts-up. The VR provides a locally unique name for each client and periodically checks registered entities to ensure their liveliness, removing stale entries if necessary.

The VR consists of the following components, shown in Fig. 5:

- The transport handler allows different protocols to be used between Tycho producers, consumers, mediators, and the VR. Currently, TCP sockets, HTTP, and Internet Relay Chat are supported.
- The local store provides an abstract interface to a mediator's information store. The store itself can be implemented using a variety of data storage technologies.

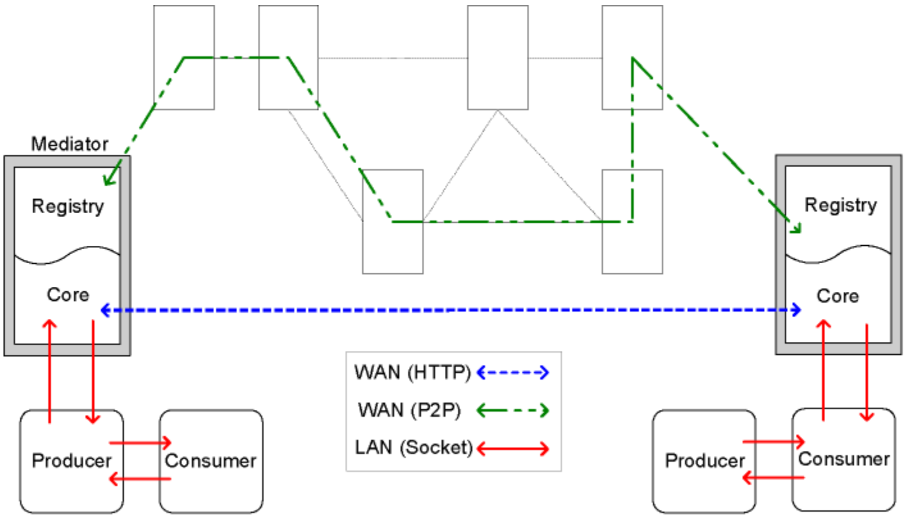


Fig. 4 The architecture of Tycho

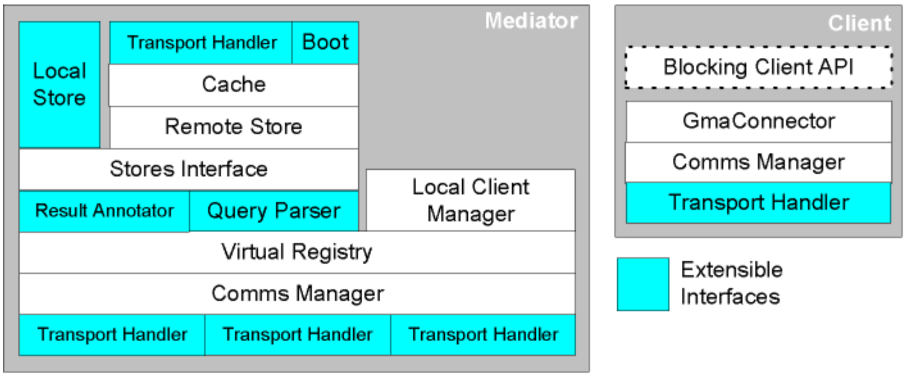


Fig. 5 A layered view of Tycho's architecture

Currently Tycho provides a JDBC-based storage medium and an in-memory data structure (simple store). JDBC permits the use of a range of SQL storage technologies ranging from Oracle to MySQL. The in-memory implementation is provided to simplify the deployment, in situations where a JDBC-based database is unavailable.

- The query parser and result annotator components translate queries and responses into an intermediate internal format in order to allow Tycho to support different query languages and permit interoperability with other systems in the future. Tycho currently supports a subset of the ANSI-SQL query language and LDIF [13] as a response mark up.

Tycho's VR provides:

- Information for uniquely identifying a client,
- URLs that are used by the transport handlers to locate and communicate with a client,
- A schema field, which can be used to store information about the capabilities of a producer or consumer.

When a mediator receives a query from a client, it performs a look up against locally registered entities, and then potentially dispatches the query to the rest of the VR. The local mediator requires a bootstrap service for locating other mediators and a transport handler for dispatching queries. Together, these services are supported by two protocols:

- HTTP: This interconnect uses a well-known server to maintain a list of existing mediators within the VR, which it uses to bootstrap a new mediator. Inter-mediator communication is provided by the HTTP transport handler, which dispatches queries to remote mediators by sending messages serially to all mediators. Currently, placing Tycho behind a proxy server configured to require authentication can restrict access to the HTTP-interconnect.
- IRC: This inter-connect uses a dynamic discovery process based on Internet Relay Chat.

The HTTP connect is one that is commonly used; a novel alternative is the IRC VR-interconnect, discussed next.

3.1 IRC VR-interconnect

IRC networks [14] typically have groups of servers connected in a graph topology, which can be configured to route messages and provide fault tolerant capabilities. For example, QuakeNet [15] can support many hundreds of thousands of clients simultaneously [16]. The IRC DNS servers can be bound to a pool of IRC servers to provide load balancing based on server load and geographic location. A DNS query will then respond with the address of a 'suitable' lightly loaded server. Alternatively, by using a database of IP address prefixes, it can provide the address of a server that is geographically close to the client. In the event that a server becomes unavailable, DNS can be used to direct a client to available servers.

Tycho uses the combination of DNS records and the IRC servers to bootstrap the VR. This provides the VR service with a measure of fault tolerance, as it avoids a single point of failure, provides scalability and importantly by-passes the need to install servers to provide the functionality required by Tycho. An IRC client (bot) is then used by the VR to locate and communicate with other instances of the registry within the VR.

Figure 6 illustrates the steps that a consumer goes through to discover a producer using the IRC VR.

1. When the mediator is started, an IRC bot uses DNS information to locate a server and joins
2. The IRC bot attempts to join a pre-determined IRC channel.
3. A consumer will register itself with its local mediator.
4. In this step:
   (a) The consumer sends a query for producers to its local mediator.
   (b) The IRC bot within the local mediator sends the query to the IRC server, which broadcasts it to other bots associated with remote mediators.
   (c) The query is run against the local data store at each mediator and matches are sent back over IRC and delivered to the consumer.
5. The consumer communicates with the producer via the mediator, in this case using a combination of sockets and HTTP.

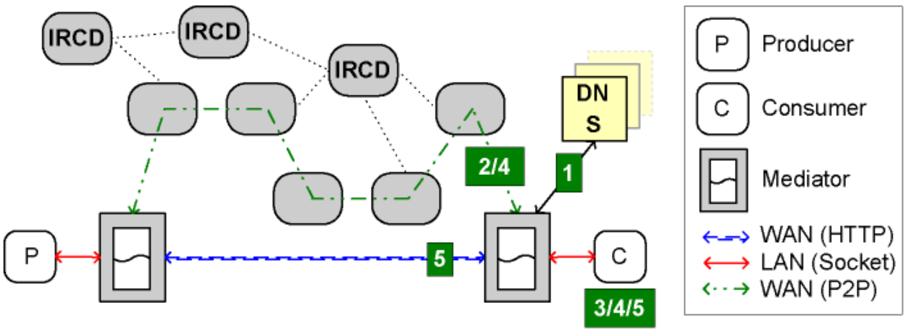


Fig. 6 The steps a consumer takes to discover a producer using the IRC VR

IRC servers can be configured to use encryption to protect messages while in transit over the Internet. If a public IRC network is used for Tycho there are various services provided to help prevent unauthorised access to the VR bots. The channel through which the bots communicate can be password protected and bots can communicate using ‘private messages’, which the IRC network does not expose to other parties. Higher levels of security can be provided with a private IRC network, which allows the maximum amount of control over the security of the system, although it also adds the administrative overhead of maintaining an IRC network.

3.2 Security

Security is an essential requirement for any distributed system. Tycho’s architecture is designed to support both encryption and access control to provide a secure environment. Encryption is provided at the transport handler level using SSL to encrypt messages sent via the HTTP, Socket and IRC handlers. Access control is provided using a layered approach. In keeping with the design philosophy of Tycho, we re-use existing infrastructure. Access control is can be via the use of a proxy server, or the security features of an IRC daemon. For instance, when deploying Tycho on a cluster a common configuration uses a proxy server on the head node to control access to mediators running on compute nodes. An alternative mechanism for access control is provided by a pluggable authentication library, which could interface with existing security protocols and solutions such as WS-Security or the Java Authentication and Authorization Service (JAAS).

4 Performance tests

A performance study of Tycho against similar systems has been made. For the purposes of evaluating Tycho’s messaging performance, comparative tests were made with the NaradaBrokering system the performance of Tycho’s virtual registry was compared to Globus MDS4 and to R-GMA.

4.1 Messaging performance: Tycho versus NaradaBrokering

Figure 7 shows two NaradaBrokering clients communicating over the Internet. NaradaBrokering does not allow clients to communicate directly; at least one broker is always required. This differs from Tycho in which messages are only routed via

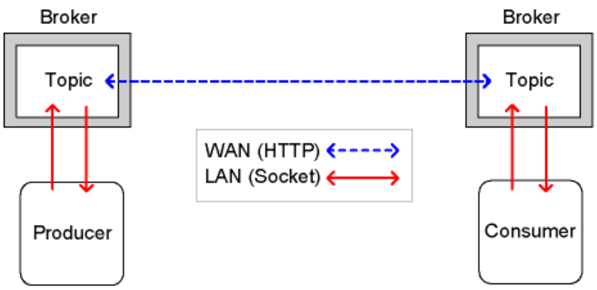


Fig. 7 A high-level view of the NaradaBrokering architecture

Table 1 System configuration

Processor type	Dual Xeon (Prestonia)
Processor speed	2.8 GHz
Processor cache	512 K L2 Cache
Font side bus	533 MHz
RAM	2 Gbytes ECC
Storage	80 Gbytes EIDE
JVM	Sun Java Version 1.5.0-b64
OS	Debian Linux 3.1, Kernel 2.4.32

Mediators to facilitate wide-area communications. Two test configurations were used to measure the message passing performance of Tycho and NaradaBrokering. A nine-node cluster, see Table 1 for details, was used to perform the performance tests. A Java sockets-based implementation of the ping-pong test was used to measure the baseline communication performance. By comparing the measurements taken when timing Tycho and NaradaBrokering with the baseline performance, it was possible to analyse the overhead of sending and receiving messages for each system.

4.1.1 The ping-pong tests

The first test is used to assess end-to-end performance using a traditional ping-pong benchmark to measure the round trip time to send and receive messages of varying sizes between a single producer and consumer, and can be used to assess latency and bandwidth. Two different arrangements of producers, consumers, and mediators/brokers were used to measure the latency and bandwidth for two different types of communication:

- The performance of TCP-based communications over Fast Ethernet (LAN),
- The performance of communications using a combination of TCP (LAN) and HTTP (WAN). This test measures the extra overhead of using mediators/brokers in an arrangement typical for wide-area communications.

For each test the layout of NaradaBrokering and Tycho components where matched as closely as possible in order to provide comparable results (see Fig. 8). The main difference is that NaradaBrokering does not allow for direct communication between end-points; messages must flow between clients via a broker(s), which handles the routing of messages. Conversely, clients in Tycho by-pass the mediator and undertake direct communications over the host or LAN.

Figure 8 shows Tycho and NaradaBrokering configuration for the two the Ping-Pong tests. In the first test (Fig. 8a and b) the consumer is run on one host and the producer on another, they communicate over Fast Ethernet via sockets. The Tycho consumer and producer communicate directly using sockets, only using the mediator to bootstrap the test. The NaradaBrokering consumer and producer also use sockets to send the messages, but these messages are routed via the broker.

In the second test (Fig. 8c and d) four hosts are used to allow components to be arranged in a manner that would emulate typical WAN communications. Both Tycho and NaradaBrokering components are arranged the same way: The consumer and producer communicate with two separate brokers/mediators using sockets, and the brokers/mediators communicate using HTTP.

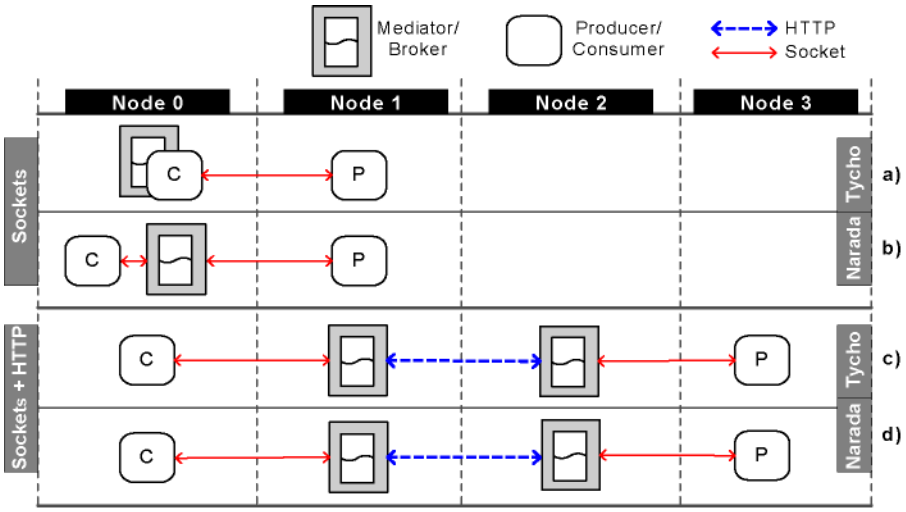
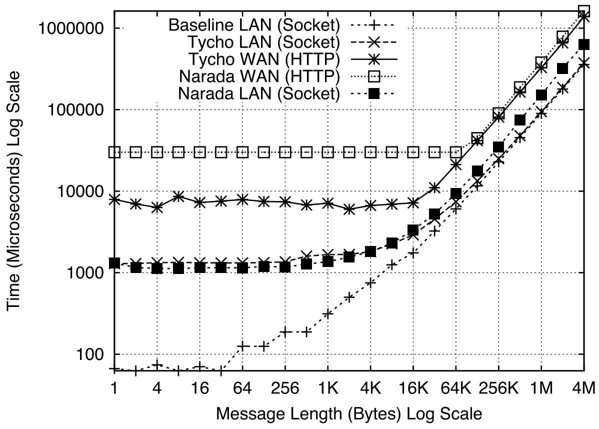


Fig. 8 Test configuration of the PingPong tests

Fig. 9 Latency results



Latency and bandwidth (LAN) The latency results are shown in Fig. 9. Baseline Java has a latency of less than 200 µs for messages <512 bytes after that it shows a linear increase with message size. For messages <128 Kbytes, both Tycho and NaradaBrokering show an additional fixed latency of around 1.2 millisecond over the baseline Java's performance. The difference between NaradaBrokering and Tycho latency is negligible until 4 Kbytes; thereafter the difference between the systems increases. At the 4 Mbyte message size, Naradabrokering is 59% slower than Tycho.

The bandwidth results are shown in Fig. 10. Baseline Java's bandwidth utilisation peaks at 11.5 Mbytes/s. The bandwidth achieved by NaradaBrokering and Tycho, for messages <8 Kbytes, is approximately 55% of that of baseline Java. After this point, Tycho's bandwidth continues to increase and plateaus 10.9 Mbytes/s; whereas NaradaBrokering peaks at 7.3 Mbytes/s. Thereafter the bandwidth achieved by NaradaBrokering gradually falls off, whereas Tycho stays approximately constant.

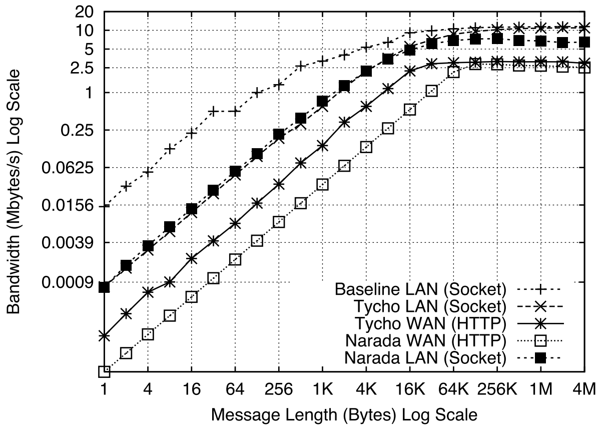


Fig. 10 Bandwidth results

Latency and bandwidth (WAN) For messages <16 Kbytes, there is a constant 5 millisecond difference in latency between Tycho and NaradaBrokering using HTTP, the latter being the slower. At the 128 Kbyte message size both systems display the same latency. The maximum bandwidth for both Tycho and NaradaBrokering using HTTP communications for messages is similar and the maximum attained is around about 26% of the maximum achievable.

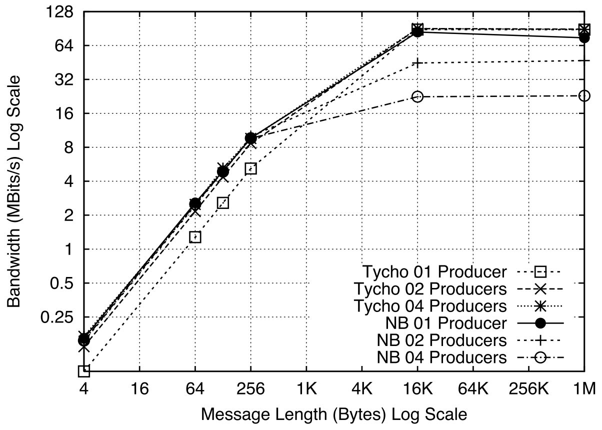
4.1.2 Scalability tests

The scalability tests were designed to measure the performance of Tycho and NaradaBrokering as the number of producers or consumers is increased. In both tests, the single consumer/producer and mediator/broker are started on separate nodes within the test cluster, with the remaining nodes used to run clients. Initial experiments showed us that fourteen clients are sufficient to saturate the Fast Ethernet network (see Fig. 11).

Many producers This test measures the affect on a consumer receiving messages from an increasing number of producers. The results from this test show the maximum number of messages per second a consumer can receive before the performance falls off.

The results from these benchmarks are shown in Fig. 11. With one consumer, and multiple producers (1–14), for messages $\leq$ 256 bytes, both Tycho and NaradaBrokering achieve a maximum bandwidth of approximately 9.5 Mbits/sec. Tycho peaks at 90.4 Mbits/sec. Whereas NaradaBrokering peaks at 84.2 Mbits/sec, with one producer, as the number of producers increases the bandwidth falls to 15.2 Mbits/sec for 14 producers.

In Tycho the bandwidth of a single producer is inhibited until message sizes are large enough to saturate the network. This is because a socket is created for each message sent and the rate that sockets can be created is limited. When more producers are used sufficient messages can be sent to saturate the receiver. NaradaBrokering reuses sockets, so it is not affected by this limit. It was observed that, with the default JVM heap size of 512 Mbytes, when using five producers, each sending 16 Kbytes



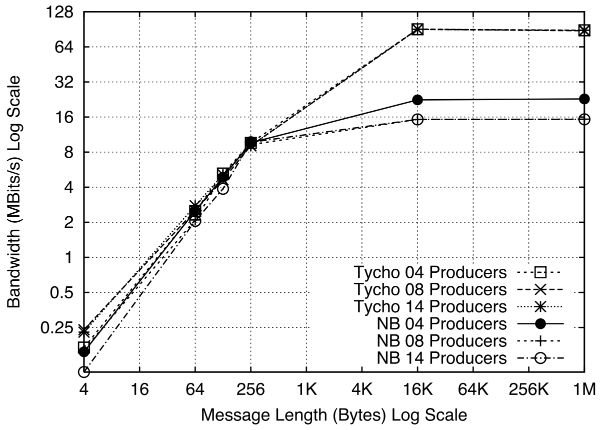


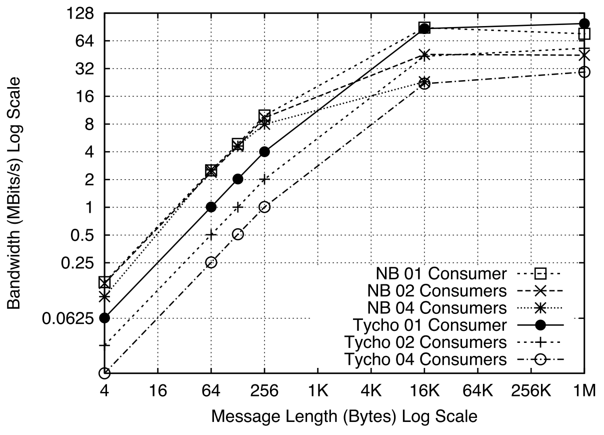
Fig. 11 Many producers (1–14)—single consumer, bandwidth versus message length

messages that NaradaBrokering runs out of heap space. By increasing the maximum heap size up to 1.5 Gbytes the tests could be completed. The Tycho heap size was left as the default.

Many consumers This test measures the performance of a single producer sending messages to varying numbers of consumers. The results from this test show how the performance of a producer is affected, as it sends messages to an increasing number of consumers.

The results from these benchmarks are shown in Fig. 12. The bandwidth of Tycho and NaradaBrokering increase steadily as the message size increases until 16 Kbytes. The bandwidth peaks at 1 Mbyte with a bandwidth of 88.5 Mbit/s. The available bandwidth is divided between the consumers, which means as the number of consumers increases the bandwidth proportionately decreases.

As with the producer tests, the heap size for NaradaBrokering had to be increased to 1.5 Gbytes to run the tests, however NaradaBrokering exhausts the heap with three



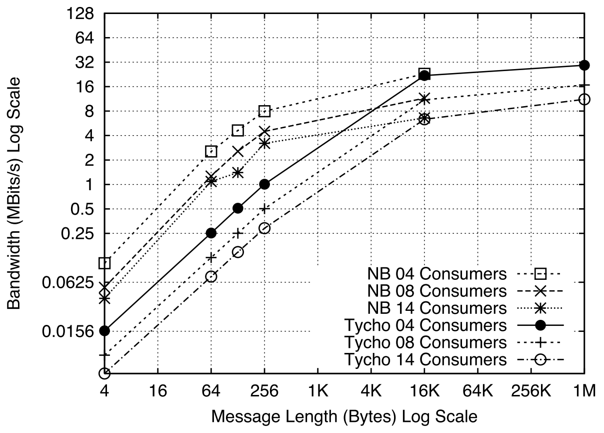


Fig. 12 Many consumers (1–14)—single producer, bandwidth versus message length

consumers receiving 1 Mbyte messages. We were unable to increase the heap further due to memory limits on the test cluster.

4.1.3 Summary

Latency and throughput When looking at end-to-end performance, on a LAN for messages less than 2 Kbytes, Tycho and NaradaBrokering have comparable performance. Here Tycho achieves 95% of the maximum bandwidth, whereas NaradaBrokering uses 65.3%. The peak bandwidth achieved by Tycho using HTTP was 27% of the maximum attainable and NaradaBrokering achieved a maximum of 26%. Overall, the performance of NaradaBrokering and Tycho is not that different. Tyco's current performance is inhibited by the fact that it creates a new socket for each message send, whereas NaradaBrokering reuses sockets instances once they have been created. Incorporating such as scheme in Tycho will further reduce its latency.

Scalability summary The scalability tests have shown Tycho and NaradaBrokering producers and consumers to be stable under heavy load although performance is weaker when there is a large ratio of consumers to producers. The heap size for NaradaBrokering becomes a limiting factor in circumstances where a broker is receiving messages faster than it can send them, as the internal message buffer fills until the heap is consumed. This problem could be avoided by implementing a flow control mechanism. The Tycho tests were performed without modifying the heap size, this is because the way it is implemented prevents messages from being received faster than they can be sent, as there is limited buffering.

4.2 Registry performance tests: Tycho compared to R-GMA and MDS4

We tested Tycho against R-GMA and MDS4 in order to show that our philosophy of keeping the core functionality as simple as possible yields performance gains over these systems while still supporting the registry functionality required. In Tycho, more complex functionality is added to the edge of the implementation rather than by increasing the complexity of the core, thus is it is essential that the core perform well.

Tycho, MDS4 and R-GMA all use different terminology to describe the same functional components. In the following sections we use the label 'registry' to refer the collection of services each system uses to provide registry functionality. For Tycho this is the mediator, for MDS4, the container running the services and for R-GMA the Tomcat container running the registry and schema servlets. We call programs interacting with the registry 'clients'.

For the tests we created a set of randomly generated strings to act as attributes for records to be inserted into the registries. A single record, with no mark up, had an average size of 114 bytes. Two tests were used to assess the performance of the registries.

- The first test [**S1**] simulates a client searching the registry for records matching some known attributes. Systematic queries are generated using a function to select a record name at random from the test data to guarantee the query will only match one record. In Tycho this is specified using the following `SQL: SELECT * FROM clients WHERE name='randname';`
- The second test [**S2**] measures the worst-case scenario of the client requesting all of the records from within the registry. In Tycho this query is specified using the following `SQL: SELECT * FROM clients;`

By configuring the Tycho core VR services, described in Sect. 3.1, and arranging these components in different ways we have been able to test the performance of the Tycho's VR under variety of different circumstances and compare it to the performance or MDS4 and R-GMA. Figure 13 shows the configuration for each Test 1 and 2.

The cluster detailed in Table 1 was used for these tests. Sun's JVM version 1.5.0-b64 was used for Tycho (0.7.3) and MDS4 (Globus 4.0.1), R-GMA (gLite 1.5) requires Sun JVM 1.4.2 06. All three systems were run with their security features disabled.

In Test 2 the clients and registries were evenly distributed using a round-robin approach between the compute nodes. The cluster was monitored using Ganglia to provide information such as memory use. Each test was repeated 10,000 times and the

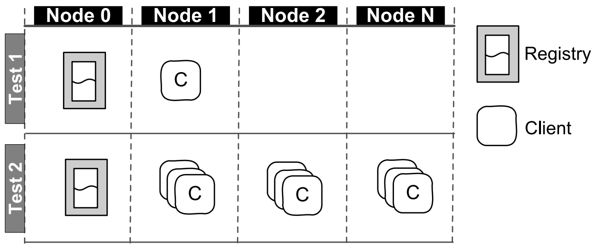


Fig. 13 Configuration of Test 1 and Test 2

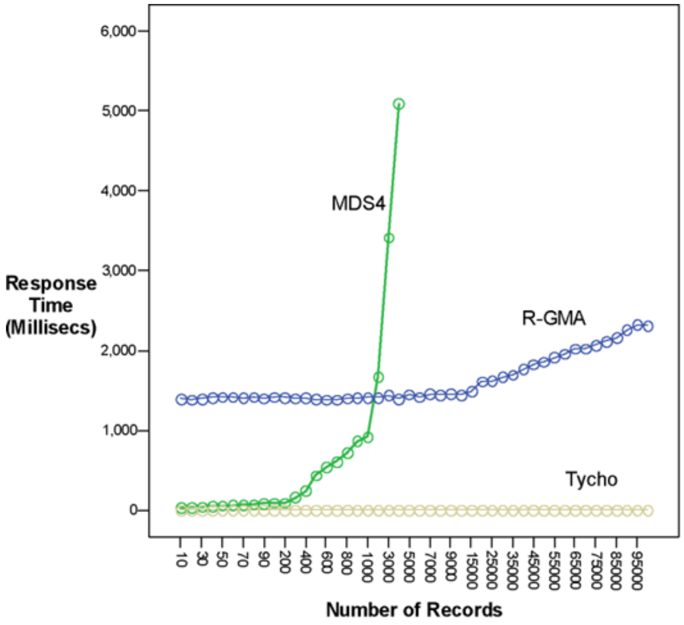


Fig. 14 Query response time versus the number of records for query when selecting a single random record from the registry (S1)

benchmarks were written so that the time to bootstrap the test did not interfere with the measurements being gathered. For Tycho, the JVM was invoked with no command line options, for MDS4 and R-GMA when the default heap size was exhausted the tests were repeated using up to a maximum heap of size 1.5 Gbytes.

4.2.1 Test 1

This test simulates a client searching the registry for records matching some known attributes. The number of records published into a single registry on one cluster node was varied from 10 to 100,000. A client on a second cluster node queried the registry using the two different types of queries (**S1** and **S2**). The test aims to measure the performance of the different registries with increasing numbers of records. The time to complete each query was recorded.

In Fig. 14 a single random record is being selected from a registry (**S1**), while the number of record in the registry is increased. Tycho has a constant response time of

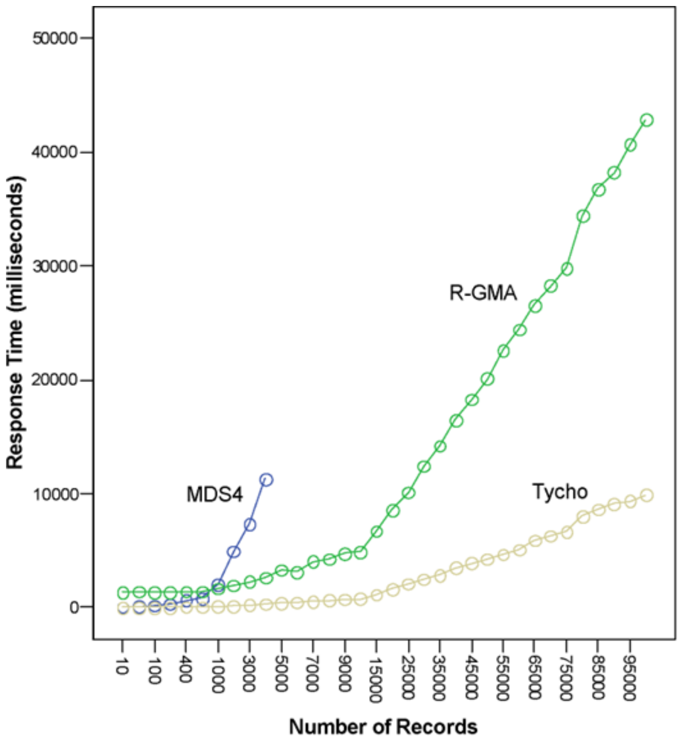


Fig. 15 Query response time versus the number of records for a query that selects all the records from the registry (S2)

3.5 ms; for up to 100,000 records HSQLDB has a constant overhead for this simple query. R-GMA has a fixed query time of 1400 ms up to 20,000 records at which point it increases steadily to 2314 ms for 100,000 records. Unlike R-GMA and Tycho, MDS4 response time increases with every extra record. It starts with a query time of 21.06 ms for 10 records and increases steadily to 5095 ms for 4000 records. After 4000 records the Globus container executing the MDS4 services gave an out of heap error with the heap size set to the maximum supported by the test hardware of 1.5 Gbytes.

The results in Fig. 15 uses the same registry data as the previous test, but the query executed selects all of the records in the registry (**S2**). Both Tycho and MDS4 have a constant increase in response time in relation to the number of records with Tycho increasing by around 0.08 ms per record and MDS4 1.93 ms. R-GMA has a constant response time of approximately 1462 ms for up to 2000 records, this can probably be attributed to some kind of fixed cost in the implementation. After this point the response time increases steadily up to 42,871 ms for 100,000 records, which is 32,973 ms more than Tycho. It is interesting to note that the curves for MDS4, for both types of query, track each other closely. The difference increases from 71.6 ms for 100 records, up to 6208 ms for 4000 records. We believe this is because the network latency has a greater impact for the larger response size associated with the select all query.

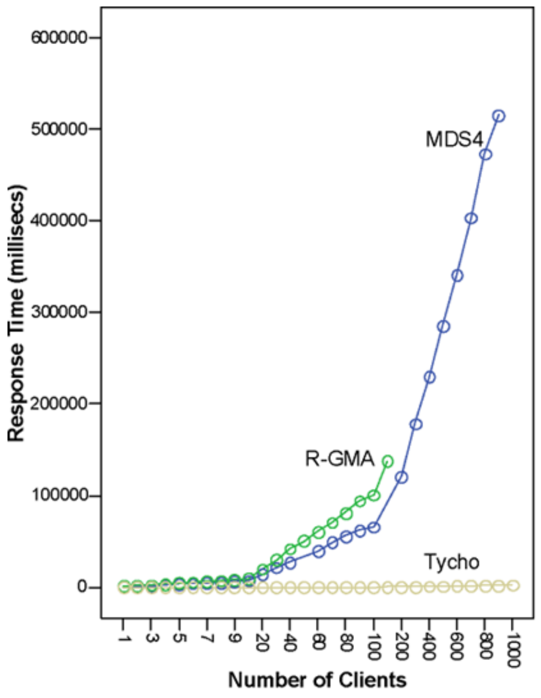


Fig. 16 Query response time versus the number of clients querying a single registry instance

4.2.2 Test 2

In this test the three registries were loaded with 1000 records and the number of clients performing simultaneous queries was varied from 1–1000. The registry was run on one node and the clients were evenly distributed amongst the eight other nodes. This test measured the effect of increasing the number of simultaneous queries on a single registry; it attempts to show how well a single registry copes with increasing numbers of local clients. The time to complete a query was measured.

Figure 16 shows the relationship between the query response times as the number of concurrent clients that query a single registry increases. For Tycho the response time increases by on average 2.3 ms per additional client. The response time for MDS4 on average 14,077 ms higher than Tycho, and for each additional client costs approximately 655 ms. R-GMA has an added 20,094 ms latency over Tycho for the same number of clients (6016 ms higher than MDS4) with 989 ms added to the response for every additional concurrent client. When testing R-GMA, after 150 clients the registry servlet crashed with an out of stack memory error that prevented us from completing the tests up to 1000 clients. We believe this is due to the rapid creation and destruction of the R-GMA consumers, the test code selects a random record per iteration and in R-GMA a new consumer must be created to run a different query.

Registry summary When testing the affect of the number of records on response time, we see that when selecting a single record from 100,000, Tycho responds 32 seconds faster than R-GMA. MDS4 runs out of heap space for larger records sizes, which suggests that they should look at either storing the data more efficiently or moving to a file-backed store that is not limited by heap size. The results of testing the Tycho stores using HSQLDB, MySQL and the Simple Java store show that HSQLDB performs best (see [16] for details). Perhaps R-GMA should consider using HSQLDB instead of MySQL too.

In the multiple client tests, Tycho's VR had a lower response latency than R-GMA and MDS4. With 100 clients Tycho was 94 seconds faster than R-GMA and 65 seconds faster than MDS4. The results highlight that one of the strengths of our implementation is its performance under load. Tycho's performance is linear with regard to both increasing numbers of clients and response sizes. MDS4 uses a global schema that must be consistent in every MDS4 instance for interoperability, this reduces its flexibility. R-GMA is the closest match to Tycho's architecture although it does not allow for the same level of flexibility with regard to the data it can store, as the schema is global to the R-GMA system and must be configured before records can be inserted.

4.2.3 Test 3

In this test a single Tycho client on one node queried the whole VR for a single record. The VR was made up of 1–1000 mediators evenly distributed on the compute nodes. Each mediator contained 1000 records so as the number of mediators increased so did the total size of the VR, reaching a maximum of 100,000 total records when using 1000 mediators. This test measures the performance of the Tycho VR as the number of mediators in the system is increased. We tested the two different VR interconnect protocols, IRC and HTTP, with caching on and off. The average response time for a query was measured.

Figure 17 shows the configuration of Test 3—in this test the consumer and the IRC daemon where located on one node, and the registry/mediator instances were gradually increased in a round-robin fashion on the other nodes. Figure 18 shows the affect on response time as the number of mediators and records in the VR is increased. For HTTP (no caching) an extra 1.24 ms is added per new mediator. This performance is attributed to the serial way the HTTP dispatches queries to other mediators. With IRC (no caching) until 500 mediators (50,0000 records) each extra mediator and 1000 records add on average 0.41 ms to the response time. The peak at 500 mediators for IRC (no caching) marks the point where the test cluster consumed its available RAM and started to use swap space. The marked difference between the performance of IRC and HTTP is mainly because queries are sent to the mediators in parallel by the IRC daemon as opposed to serially for HTTP. IRC (caching on) has the lowest average response time adding approximately 0.13 ms for each extra mediator.

VR interconnect summary The IRC interconnect performed better when routing queries between mediators than HTTP. For 1000 mediators, the response time was

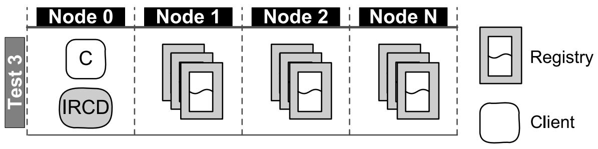


Fig. 17 Configuration of Test 3

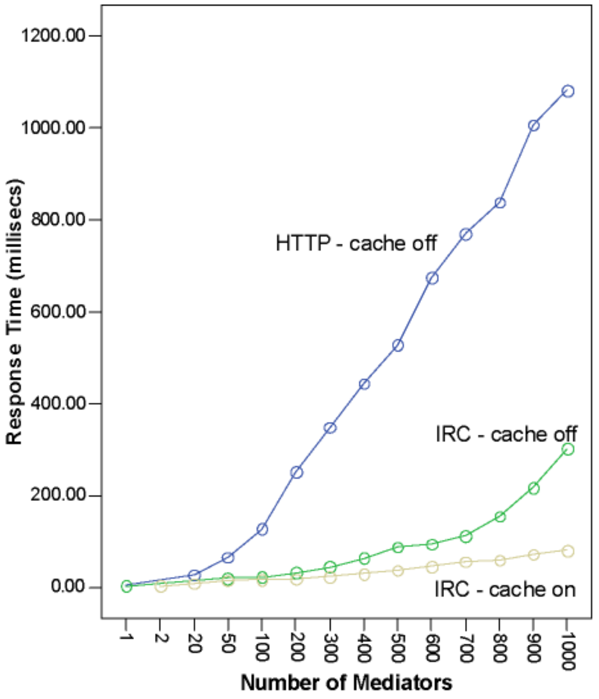


Fig. 18 Query response time versus the number of Tycho mediators

778 ms faster, and with caching on, the latency was reduced by a further 220 ms. The HTTP interconnect could be improved by sending queries to mediators in parallel and the IRC-interconnect could be further improved by using multiple IRC channels to provide a network overlay to allow more efficient message routing. The HTTP-interconnect can perform better than IRC as message response size increases. This is a result of the IRC having to fragment the response into multiple messages due to limitations of the IRC protocol. One solution to overcome these issues would be to use a hybrid VR-interconnect using a combination of transport handlers to exploit the strengths of the different approaches, i.e. using IRC to route queries and HTTP to deliver responses over a certain size.

5 Summary and conclusions

5.1 Summary

In this paper we first outline our needs and the motivation for developing a system that provides integrated asynchronous messaging and a distributed registry. The architecture and main features of the most popular systems used for messaging (NaradaBrokering) and as registries (MDS4 and R-GMA) were discussed. We then described architecture and features of Tycho, our implementation of a wide-area asynchronous messaging system that includes an integrated distributed registry. In order to justify the use of Tycho we then presented extensive performance tests against NaradaBrokering, MDS4 and R-GMA. These tests show that Tycho has comparable, if not better, performance and capabilities then these more matures system.

5.2 Conclusions

We designed Tycho to have a relatively small, simple and efficient core, so that it has a minimal memory footprint, is easy to install, and is capable of providing robust and reliable services. More sophisticated services can then be built on this core and are provided via libraries and tools to applications. This provides us with a flexible and extensible framework where it is possible to incorporate additional feature and functionality, which are created as producers or consumers, and do not affect the core. Tycho's functionality has all been incorporated within a single Java JAR and requires only Java 1.5 JDK for building and running applications.

Tycho performance is comparable to that of NaradaBrokering, a more mature system. Certain features of NaradaBrokering are superior to those of Tycho, but its memory utilisation and indirect communications are limiting features. Whereas, compared to MDS4 and R-GMA, Tycho shows superior performance and scalability to both these systems. In addition, we would argue that both MDS4 and R-GMA have problems with memory utilisation and without significant extra effort limited scalability.

NaradaBrokering and R-GMA currently have a richer functionality and features than Tycho, including support for various Web Services specifications, and Grid APIs. Additional APIs and specifications can be easily incorporated into Tycho by simply creating compliant producers and consumers. An important advantage of Tycho's architecture is that addition of further producers/consumers will not affect its core, or existing producers/consumers.

5.3 Future work

Tycho is being used in several projects including GridRM and the Semantic Log Analyser [17] project at Portsmouth. In addition, Tycho is being used to provide service discovery for the VOTechBroker [18], which is part of the European Virtual Observatory [19] project. The Virtual Observatory allows astronomers global access via a web portal to various astronomical data archives. The VOTechBroker facilitates the use of existing infrastructure to execute jobs submitted through a web interface. Participating sites publish capabilities, such as the batch submission details via a Tycho producer; the VOTechBroker uses a Tycho consumer to discover the remote resources and uses the capabilities published via Tycho to select a site to submit the jobs. We expect the process of integrating Tycho with other systems such as the VOTechBroker will lead to the development of more sophisticated tools and services, such as aggregate and multi-threaded producers.

Even though Tycho's registry performance is better then both MDS4 and R-GMA, there are still a number of areas that we feel could be improved. One way to improve performance is altering caching in the mediator to include local data-store queries in addition to remote responses. In addition, the message-passing performance could be improved by changing the socket transport handler to use thread pooling to further reduce the cost of sending messages. In the future, we will add functionality into Tycho to provide services that are more advanced. One key area is to develop transport handlers that support SSL sockets or HTTPS to provide secure communication. Other features may include support for transactions, various Web Services specification, for example WS-notification, and producers/consumers that are suitable for computational steering.

Tycho [20] is currently available as a binary release to developers interested in investigating and further enhancing its capabilities.

References

1. SOAP. http://www.w3.org/TR/soap/
2. GridFTP. http://www.globus.org/toolkit/docs/4.0/data/gridftp/
3. UDDI. http://www.oasis-open.org/committees/uddi-spec
4. OpenLDAP. http://www.openldap.org/
5. Baker MA, Smith G (2003) GridRM: an extensible resource management system. In: Proceeding of the IEEE international conference on cluster computing (Cluster 2003), Hong Kong, 1–4 December 2003. IEEE Computer Society Press, ISBN 0-7695-2066-9, http://gridrm.org
6. Pollickara S, Fox G, Gadgil H (2005) On the creation and discovery of topics in distributed publish/subscribe systems. In: Proceedings of the IEEE/ACM grid 2005 workshop, Seattle, WA, pp 25–32
7. Schopf J et al (2005) Monitoring and discovery in a web services framework: functionality and performance of the globus toolkit's MDS4, ANL Tech Report ANL/MCS-P1248-0405, April 2005, http://www-unix.globus.org/toolkit/mds/
8. Cooke AW et al (2004) The relational grid monitoring architecture: mediating information about the grid. J Grid Comput 2(4). http://www.r-gma.org/
9. GMA. http://www-didc.lbl.gov/GGF-PERF/GMA-WG/
10. DataGrid. http://eu-datagrid.web.cern.ch/eu-datagrid/
11. EGEE. http://public.eu-egee.org/
12. Pallickara S, Fox G (2003) NaradaBrokering: a middleware framework and architecture for enabling durable peer-to-peer grids. In: Proceedings of ACM/IFIP/USENIX international middleware conference middleware-2003, lecture notes in computer science 2672. Springer, pp 41–61, ISBN 3-540-40317-5. http://www.naradabrokering.org/
13. RFC 2849—The LDAP Data Interchange Format (LDIF). http://www.faqs.org/rfcs/rfc2849.html
14. RFC 1459—Internet Relay Chat Protocol. http://www.faqs.org/rfcs/rfc1459.html
15. QuakeNet. http://irc.netsplit.de/networks/
16. Baker MA, Grove M (2006) A virtual registry for wide-area messaging. In: Proceeding of the IEEE international conference on cluster computing (Cluster 2006), Barcelona, Spain, September, 2006, ISBN: 1-4244-0328-6
17. Slogger. http://dsg.port.ac.uk/projects/slogger/
18. VOTechBroker. http://dsg.port.ac.uk/projects/votb/
19. European Virtual Observatory. http://euro-vo.org/
20. Tycho: a resource discovery framework and messaging system for distributed applications. http://acet.rdg.ac.uk/projects/tycho/

Mark Baker is a Research Professor of Computer Science at the University of Reading in the School of Systems Engineering. Mark is involved in the research and development of various middleware technologies for Clusters, the Grid, and Wireless Sensors Networks. Mark has published widely in journals and conferences on his research interests, which include all aspects of distributed systems. Mark is a senior member of the IEEE Computer Society and is involved many IEEE activities and events.

Mat Grove is a Post Doctoral Research Assistant within the Centre for Advanced Computing and Emerging Technologies (ACET) at the University of Reading in the School of Systems Engineering. Mat's research interests include messaging in distributed systems, wide-area systems monitoring, server virtualisation and wireless sensor network technologies.