

COMP9243 — Week 8 (12s1)

Ihor Kuz, Manuel M. T. Chakravarty

Middleware

Middleware consists of a layer of services added between those of a regular network OS and the actual applications. These services facilitate the implementation of distributed applications and attempt to hide the heterogeneity of the underlying system architectures (both hardware and software). Middleware is usually based on a particular *paradigm*, or model, for describing distribution and communication. Providing such a paradigm automatically provides an abstraction for programmers to follow, and provides direction for how to design and set up the distributed applications.

Recall from the first lecture that there are several advantages and disadvantages to the middleware approach. The greatest advantage is that it runs in user space and can provide independence from OS, network protocol, programming language, etc. Unfortunately, this independence means that a lot of standard OS services must be reimplemented by the middleware, leading to an overabundance of features and bloat. One direction that current middleware is taking is to increase flexibility (and reduce bloat) by focusing on adaptation and reflection.

The most popular (and widely used) middleware abstraction is that of distributed objects. Besides distributed objects, other abstractions include RPC, transaction processing, message passing, events, shared data spaces, and web services. In this lecture we focus on distributed object-based middleware, message-oriented middleware, and (event-based) publish/subscribe middleware.

Distributed-Object Middleware

Recall from the second lecture that *remote method invocation (RMI)* is an extension of the RPC model of communication to objects. Whereas RPC involves the invocation of procedures on remote servers, RMI involves the invocation of methods on remote objects. In this lecture we will look at *distributed objects*, the distributed system paradigm based on using RMI to provide communication and transparency.

The main advantage of RMI over RPC is the fact that it combines state and operations on that state into a single entity, an object. Thus, whereas operations in RPC are directly associated with a server, operations in RMI are only associated with an object. This means that the actual location of the object (e.g., the server on which it is hosted) is transparent to the client invoking its methods. This transparency applies not only to the object's location, but also encompasses replication and migration transparency. Encapsulating state in an object also allows the (server) programmer to keep track of related state more easily than in the RPC model. Similarly the distributed object model allows concurrent access to related state to be synchronised more easily (that is, with less programmer effort) than with the RPC model.

The popularity of the distributed-object model is largely due to the fact that there is a natural mapping between the common distributed-systems model of communicating entities (e.g., client and server, resource and user, etc.) and the distributed-object model of communicating objects. By encapsulating state and operations on that state, objects also provide a natural unit of replication and migration. Furthermore, object boundaries also provide a good place to impose security restrictions and perform auditing. An object-based model also provides a familiar abstraction to programmers and allows the use of many existing object-based tools and techniques for designing systems and applications.

Challenges

The main challenge in designing a distributed-object based middleware system is to provide transparency. The model naturally provides location, migration, and replication transparency. A greater challenge is to provide failure transparency. As discussed previously, remote method invocation can fail in ways that local method invocation cannot. Moreover, trying to mask the possibility of these kinds of failures is not always a good idea. A distributed object system must, however, deal with these kinds of failures. In particular the system must deal with *partial failures*, that is failures of some components of the system (e.g., network, a server, etc.) that make it unclear where the fault occurred (for example, a remote server failure may be indistinguishable from a network failure: did the client not receive a response because the server failed, because the request never arrived at the server, or because the response never arrived at the client?). See [WWWK94] for more details.

Another main challenge of designing a distributed-object system is scalability: making sure that the system can deal with growth. In particular the system must deal with growth in the number of clients an object has, the number of objects deployed in the system, and the distance between clients and objects. Performance and flexibility are also important challenges.

When designing distributed-object based applications it is also important that the distributed nature of the application be taken into account from the beginning. Due to the communication delays and possibilities for partial failures, as well as scalability concerns, taking a non-distributed object-based design and implementing it with distributed objects retrospectively is a recipe for failure.

Architectural Model

A typical distributed object architecture is presented in Figure 1. In this architecture a client process invokes methods on a *proxy* object in its address space. The proxy object marshals the invocation and with the help of the runtime system sends the invocation request on to the *object server*. At the server the runtime system dispatches the request to an appropriate object *skeleton* who is responsible for unmarshaling the request and invoking the appropriate methods on a local object instance. Results are returned in a similar fashion.

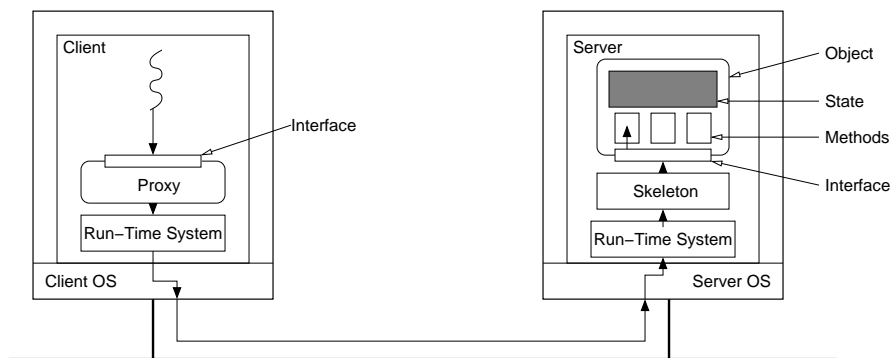


Figure 1: A typical distributed object architecture

Object Model

A distributed-object system typically distinguishes between *classes* and *objects*, where a class defines a type, and an object is an instance of a particular class. A class generally defines an *interface* that specifies the methods that objects of that class will implement. An object is referred to using an *object reference (OR)*. There are two types of object references: local references, which are valid within a single address space, and global (or remote) references, which are valid across

address spaces and machines. Most distributed object systems provide *passive objects*. These are objects that have no own thread of control associated with them. Some systems also provide *active objects*, which do have their own thread of control.

Most systems offer both persistent and transient objects. *Transient objects* are those that exist only as long as they are referenced by some client (and the object server that they are hosted on exists), *persistent objects*, on the other hand, exist until they are explicitly destroyed (even surviving restarts of their hosting object server).

Distributed-object systems offer two styles of method invocation. With *static method invocation* proxy object code is compiled (or linked) into the client code and methods are invoked on the proxy as normal. At the server side, the object server has the skeleton and object code compiled in and skeleton and object invocations proceed normally. With *dynamic method invocation* the client does not have proxy code compiled in and must create invocation requests dynamically. This involves determining the object interface at runtime and using this information to build an invocation request. At the server side the skeleton invocation must similarly be created at runtime using information about the actual object implementation.

Interfaces

A distributed object's functionality is defined by the interfaces that it exports (i.e., provides to clients). This interface defines the methods that an object implements. In many systems an object may export more than one interface, this ability is generally provided through composition of interfaces. Some systems also allow inheritance of interfaces. Because interfaces can evolve over time many systems allow interfaces to be explicitly versioned. Note that interfaces generally define only the syntax of method invocations and not the semantics.

As with RPC, interfaces are defined using an *interface definition language (IDL)*. The IDL for distributed-object systems is usually richer than that for RPC system, in particular it allows the specification of classes (i.e., types of objects), class attributes, public methods, exceptions, and other data types (for example, arrays, records, etc.).

Client Side

In order to use a distributed object a client must first bind to that object. This results in the creation of a proxy object in the client's address space. After binding, the client can perform regular local method invocations on the new proxy object. The proxy is responsible for marshaling the invocations and sending them off to the remote object server hosting the object that the proxy is bound to. In this way, a proxy is equivalent to an RPC stub that contains extra information about the location of the destination object server and object. A client must use a different proxy object for every separate distributed object that it wishes to access. Proxy object code is usually generated from an IDL interface specification and linked in with the client code. As mentioned, dynamic method invocation makes it possible for clients to invoke methods on objects whose interfaces are not known at compile time. The runtime system (RTS) at the client side provides any services necessary for the proxy to communicate with the object server. The RTS also provides some services directly to the client. These include operations for initialising the system, operations for binding to objects, and operations for manipulating object references (e.g., translating from local to global and vice versa).

Server Side

The object server's main task is to host object implementations and to allow remote invocation of methods. As mentioned earlier, an object contains state and implements methods that operate on that state. An object that implements the methods defined by a particular interface is said to implement or export that interface. An object implementation is accessed through a skeleton, which is similar to an RPC server stub. Like proxy code, skeleton code is usually generated from an IDL interface definition and linked into the object server (along with the object implementation

code). Through dynamic skeletons it is also possible for an object server to host objects whose interfaces were not known at compile time. The RTS at the server side is responsible for receiving method invocation requests from clients and dispatching them to the appropriate skeleton. The RTS is also responsible for implementing a server's invocation policies. These are policies that determine how a method invocation takes place. Examples of policies include single-threaded invocation, multi-threaded invocation, transient object invocation, persistent object invocation, dynamic object creation, etc. The tasks of the RTS are often split up into a core RTS and an *object adapter*. The core RTS takes care of communication and basic dispatching, while the object adapter takes care of invocation policies, dynamic skeleton invocation, object creation, dispatching to specific objects, etc. The object adapter is also responsible for providing object wrappers for legacy code (i.e., programs that were not built to work with a particular distributed-object system).

Naming and Binding

A local object reference is usually implemented as a local pointer to a proxy object. A remote object reference, on the other hand, must include enough information to allow clients in any address space to create a proxy and allow that proxy to find and connect to the appropriate object server hosting the object implementation. There are many different ways to implement remote ORs. In one implementation, a remote OR may, for example, contain a server address, object id, and a reference to a proxy object implementation. In another implementation, however, it may contain an abstract reference to the object which must be resolved by a separate service before being useful. An example of such a reference is a human friendly name or a globally unique number. An important characteristic of a remote OR is that it can be sent to, and used by, other processes.

In order to use a remote OR, a client must first bind to the object referenced by it. Conceptually, the binding operation maps a remote OR to a local OR. In practice this involves the creation of a proxy object in the client's address space and the locating of the object server hosting the referenced object.

Most distributed object systems provide both ORs and *names*, which are human friendly references to objects (i.e., they can easily be read and remembered by humans). In order to use a name to bind to an object, that name must first be resolved into an OR. This name resolution is generally performed by a separate name service. Note that named objects are generally also persistent objects.

Remote Method Invocation

The standard approach to remote method invocations in distributed object systems is to provide synchronous invocations. A synchronous RMI proceeds similar to a synchronous RPC. One difference between RPC and RMI is that it may be necessary for the object adapter to first create an instance of the object before the method invocation can proceed at the object server. Some distributed-object systems (including CORBA) also provide alternative RMI invocation models, including asynchronous invocations, deferred synchronous invocations, invocations on persistent objects, as well as event and callback based invocations. In the latter two, invocations are performed on functions provided by the client, rather than on the object.

Remote method invocation provides an abstraction of communication for users of distributed-object systems. While this shields users from the various underlying communication concerns, communication remains an important concern for the designers and implementors of distributed-object systems. Generally distributed-object systems are implemented according to the client-server model. Furthermore they typically make use of reliable and connection oriented point-to-point communication. Some distributed object systems also make use of group communication.

Because of RMI's remote character, compounded with the possibilities of partial failure, different RMI implementations may provide different invocation semantics (i.e., guarantees about how often a method will actually be invoked on the object at the object server) in the face of failure. A method invoked on a local object is always executed exactly once. However, due to the

possibilities of failures, retransmissions of message, and loss of replies, such a guarantee cannot be made for remote method invocations. Overall, there are four types of invocation semantics:

Exactly-once: This is the optimal case as realised by a local invocation.

At-most-once: If the invoker receives a result, the method was executed exactly once; if the invoker catches an exception, the method was executed either once or not at all. (This is the best, and most expensive, that we can do in the distributed case.)

At-least-once: If the invoker receives a result, the method was executed at least once; if the invoker catches an exception, it cannot make any assumptions. (This is of particular interest for *idempotent methods*.)

Maybe: The invoker cannot make any assumptions.

The following table provides common scenarios for the approaches to dealing with lost messages:

<i>Retransmit Request</i>	<i>Fault Tolerance Measure</i>		<i>Invocation Semantics</i>
	<i>Filter Duplicates</i>	<i>Re-execute, or Re-reply</i>	
No	n/a	n/a	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

Services

Besides an object model and the facilities for remote method invocation, distributed-object systems also provide a host of services to make the building of distributed applications easier. These services include:

Naming & Location: Services for mapping between names and object references and resolving remote object references to locations (i.e., addresses).

Directory: Services for finding objects that provide required services.

Concurrency: Services for providing concurrency control (e.g., implementing locks that span method invocations on several objects).

Event Notification: Services that provide event notification to clients and objects.

Resource Management: Services that can be used to manage distributed resources. This includes distributed garbage collection as well as the management of persistent objects.

Transaction: Services that provide the possibility to group method invocations into transaction that can possibly be rolled back on failures.

Fault Tolerance: Services that help objects and clients deal with possible failures. This may include services that replicate state for safe keeping, as well as providing exception handling and propagation mechanisms, and failure recovery mechanisms.

Security: Services that help implement secure objects. This includes services that provide authentication of communicating parties, authorisation and access control, integrity of communication and verification of data, auditing and logging of events, nonrepudiation (so that a client or object cannot deny having performed particular actions), privacy, and protection from malicious code.

The topics of concurrency control, transactions, naming, fault tolerance, and security will be handled in greater detail in future lectures.

CORBA

CORBA is an example of a widely used distributed-object system. For an overview of CORBA please see [Vin97, Vin98].

Besides CORBA, other widely used remote object systems include COM/DCOM[CHY⁺98, Rog98], .Net remoting, Java RMI[WRW96], and DCE.

Distributed Shared Objects

So far the description of distributed-object systems has assumed a *remote-object model*. In this model the object is hosted on a single object server and clients invoke methods on the object by sending invocation requests to that server. A key property of this model is that the object state is stored only at the object server, there are no copies of the state at any other servers or clients. A different approach to distributed objects is the *distributed shared object (DSO)* model. In this model an object's state can be replicated or partitioned over multiple locations. Furthermore state can be stored at both clients and object servers. Methods may be executed at any of the locations where the object's state is stored. By adding the concept of replication to the distributed-object model, the DSO model provides the benefits of greater scalability (through decentralisation and replication), but also introduces many of the problems involved with replication and consistency.

Distributed Shared Object Model

Figure 2 shows an example of a typical distributed shared object. In this DSO there are four processes bound to the object. Each process contains an instance of the object's *local representative (LR)* in its address space, and each LR exports the object interface. In this example, each of these LRs contains a copy of the object state. This example does not distinguish between clients and object servers. In the DSO model the main difference between the two is that clients invoke methods directly on the LR, while in an object server method invocations are only performed as a result of request messages received over the network.

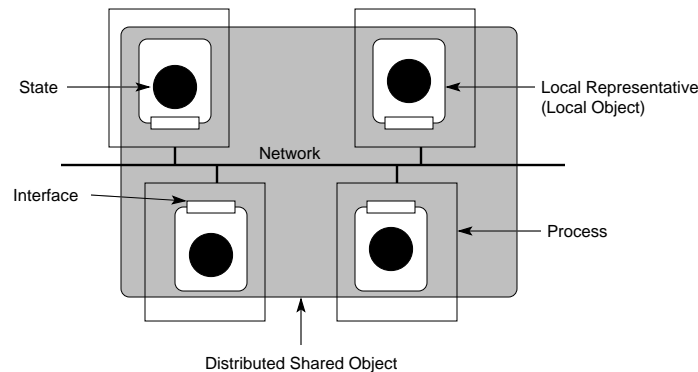


Figure 2: A distributed shared object

The DSO model distinguishes between two types of LRs, a *stateless LR* and a *stateful LR*. A stateless LR does not contain any object state, it is essentially equivalent to a proxy object. A stateful LR, on the other hand, does contain a copy of the object state (it can contain either the whole state or a part of the state). In contrast to the remote-object model, a DSO client can host either stateful or stateless LRs.

Figure 3 shows some examples of the different ways that an object's state can be distributed over its LRs. In the first example only one LR contains a copy of the state; this is equivalent to the situation in the remote-object model. In the second example the state is replicated over all LRs, that is, each LR contains a complete copy of the state. Method invocations can be performed on

any copy of the state, however, it is important that the state at the other LR's is kept consistent. In the third example, the state is partitioned over the LR's and each LR contains a different part of the state. A method invocation can be executed only at the LR's that contain the state accessed by that method. The fourth example combines replication and partitioning. In this example the state is partitioned and the partitions are replicated; two of the LR's contain copies of one part of the state, while the other two contain copies of another partition. A method invocation can be executed only at the LR's that contain the state accessed by that method. The replicated partitions must be kept consistent.

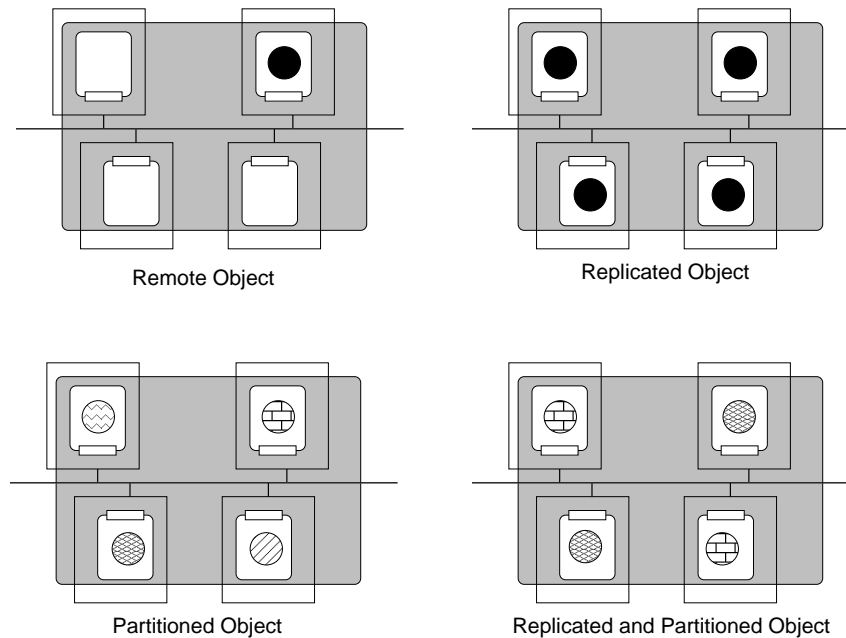


Figure 3: Examples of state distributed over a DSO's LR's

Object Server

In the DSO model the object server is a server dedicated to hosting LR's. The object server's main task is to provide an address space for LR's and provide LR's with access to local resources such as network access, local storage, etc. As with object servers in the remote-object model, DSO object servers can also provide support for dynamically creating LR's, as well as for persistent LR's. The object server may also provide mechanisms required to implement security and fault tolerance.

It is important to note that, in contrast to the remote-object model, LR's hosted by an object server do not necessarily have to be statefull. Likewise LR's hosted by clients do not necessarily have to be stateless.

Binding and Location

In order to have access to a DSO a client must bind to that object. This causes an instance of an LR to be created in the client's address space. After the LR is created it connects to (all or some of) the object's other LR's (i.e., the LR's in other address spaces). Because the object's state can be stored by many other LR's, it is not immediately clear which LR's a newly created LR should contact. Generally this decision is made with the help of an external *location service*. This is a service that maps an object reference to a particular location (e.g., network address and port). While in the remote object case mapping results in a single location, in the DSO case this mapping may result in a set of locations (for the various LR's). Another issue involved with binding

is finding the LR code to load and instantiate. This code can be compiled into the client, or may be retrieved from an *implementation repository* service. When an LR has been instantiated and has contacted appropriate other LRs it may be necessary to load a copy of the object state from some of the other LRs. Which LRs provide this state, what state the LR requires and how the state is transferred depends on the implementation of the DSO model.

Replication and Consistency

Because the DSO model allows state to be replicated the issues discussed in the previous lecture on replication and consistency play an important role in DSO systems. In particular a DSO system must provide mechanisms for keeping replicated state consistent. This means that the system (or possibly individual objects) must implement a consistency policy and a consistency protocol. There is also the issue of state migration: can LRs be migrated to different servers, and can state be migrated between different LRs? The various LRs may also take on different roles in the system. For example, some LRs can take on the roles of permanent replicas, while others take on the roles of server-initiated and client-initiated replicas. A final issue with regards to replication and DSOs is that of replicated invocations. This is a particularly relevant issue if replicated DSOs can hold and use references to other replicated DSOs.

Globe

Globe is an example of a distributed shared object system. Unlike CORBA and COM/DCOM, Globe is not a production system, but was developed as a research project to study the design and development of wide-area distributed systems. For an overview of Globe please see [SHT99]. Unlike the remote-object model, the DSO model has not been as popular. Besides Globe, other research-oriented DSO systems include Fragmented Objects [MGLNS91] based on C++ and FORMI [KDH⁺06] based on Java RMI.

Coordination-based Middleware and Publish/Subscribe

Distributed-object middleware (along with RPC, distributed files, DSM, etc.) takes a concept from non-distributed systems familiar to programmers and uses this to abstract away the intricacies of distribution. While this generally works well, there are situations in which trying to hide distribution is not a good idea and can impede scalability (as well as reliability). The category of coordination-based middleware takes the approach that the components of the system are inherently distributed, and rather than hide this, middleware should provide support for coordinating the activities of these distributed components.

Coordination-based middleware can be categorised based on the coupling between the components involved. We distinguish between whether components must be active at the same time (*temporal coupling*) and whether the components directly reference each other (*referential coupling*). The following table shows the different classes of coordination-based middleware that arise from this classification.

	<i>Temporally Coupled</i>	<i>Temporally Decoupled</i>
<i>Referentially Coupled</i>	Direct	Mailbox
<i>Referentially Decoupled</i>	Publish/Subscribe	Generative

The class of *generative* systems include tuple-based systems such as Linda and JavaSpaces. In the rest of this section we will focus on the *publish/subscribe* class of coordination-based middleware.

Publish/Subscribe

In publish/subscribe systems processes *subscribe* to messages (also called *events*) that fulfill specified criteria. When other processes in the system *publish* events, the system checks whether the events match any existing subscriptions and if so then it delivers the event to interested subscriber processes. Since publish/subscribe systems have loose coupling (that is, they are referentially decoupled) they have a high level of transparency. Theoretically the loose coupling should also provide good scalability, however, in practice it is difficult to build highly scalable publish/subscribe systems.

One of the main challenges for publish/subscribe middleware lies in how to perform the matching (or filtering) of events and subscriptions. There are two main approaches to message filtering: topic-based and content-based. In the topic-based approach all events are categorised into groups called topics. When a publisher emits an event, that event is tagged with a specific topic. Subscribers specify which topics they are interested in and subsequently receive all events related to that topic. With the content-based approach, on the other hand, there is no grouping of events. Subscriptions simply state what data are of interest and the matching process must compare the data in an event to that of a subscription to determine if there is a match. Clearly this requires more work than the topic-based approach.

Architecture

With regards to the architecture of publish/subscribe systems, there are four common models: centralised, distributed, multicast-based, and peer-to-peer. In the centralised model a single server receives both published events and subscription requests. That server is responsible for performing matching and forwarding of published events to interested subscribers. While this approach is relatively easy to implement (both for topic-based and content-based filtering) it faces scalability problems as the number of published events and the number of subscribers grows. In the distributed model broker servers create an event routing network that is responsible for sending events to the appropriate subscriber nodes. In the multicast-based model there is no central server, instead events are multicast to all nodes. Each node keeps track of its clients' subscriptions and filters events locally. This removes the burden from a single matching server (and prevents it from becoming a bottleneck), but requires a scalable multicast mechanism. Furthermore, events are sent to all nodes, even those that are not interested in them, which means that the approach scales poorly with the number of events published.

In the peer-to-peer model, the nodes form overlay routing networks so that events are sent only to those nodes that have subscribed to them. One approach for topic-based peer-to-peer systems is to use a distributed hash table (DHT) and hash topics such that they map onto particular nodes. Each topic therefore has a controlling node that is responsible for tracking subscribers interested in that topic and forwarding published events to those subscribers. Events can be multicast along a tree that is built by the event topic's controlling node as topic subscriptions are received. Implementing content-based matching in a peer-to-peer architecture is more difficult and is the subject of much current research.

Besides standard point-to-point and multicast-based approaches to communication, peer-to-peer publish/subscribe systems can also employ content-based routing to forward events. In this case the matching and routing functionality is combined and performed by router nodes. Subscription information is stored at router nodes and whenever an event arrives at such a node that node decides how to forward it based on the result of matching it with its stored subscriptions.

Replication can play a role in making the routing more efficient. If subscriptions are replicated on all router nodes, then an event can be sent to any router node, which will know which client nodes to forward it to. Of course, this means that all subscription must be replicated on all router nodes, which could get expensive. Also, the router nodes will have to keep the subscription lists synchronised. The opposite approach is to keep the subscription lists partitioned over the router nodes, which would save space and overhead of keeping replicas consistent. However, this requires events to be sent to every router node to be matched against every subscription.

Fault tolerance of publish/subscribe middleware has largely focused on using reliable multicast to deal with communication failures. However, some work has been done in tolerating process failures. One approach is to combine processes into process groups that share the same subscriptions. When a process crashes, another process from the group can take over routing of events that pass through the group. Another aspect of fault tolerance in the face of process failures is maintaining stability of routing. When a broker fails, the subscriptions that it handles (and therefore the nodes that it routes to) become inaccessible. This can be dealt with by having subscriptions expire using leases. Clients must extend leases for subscriptions in which they are interested, which means that eventually subscriptions will be re-injected into the system after they are lost due to a broker crashing. This provides self-stabilising routing.

Examples

TIB/Rendezvous [TIB05] by TIBCO is a commercial topic-based multicast-based publish/subscribe middleware. Besides basic publish/subscribe functionality TIB/Rendezvous also provides reliability and security guarantees. Java Message Service (JMS) [Sun04] specifies an API for message-oriented middleware that also provides publish/subscribe functionality. JMS is similar to JDBC in that it provides an interface specification that can be used to interface with several different implementations of the services (e.g., MQSeries, JMQ, etc.). Most implementations of JMS are based on a centralised architecture. Scribe [RKCD01] is a research publish/subscribe middleware. It is topic-based with a peer-to-peer architecture and is based on the Pastry DHT. In Scribe topics have unique identifiers that map onto nodes via the DHT. A topic node acts as the root of a topic multicast tree that is built up as subscriptions arrive.

References

- [CHY⁺98] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C.-Y. Wang, and Y. M. Wang. DCOM and CORBA side by side, step by step, and layer by layer. *C++ Report*, 10(1), January 1998.
- [KDH⁺06] Rüdiger Kapitza, Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Holger Schmidt. FORMI: Integrating adaptive fragmented objects into Java RMI. *IEEE Distributed Systems Online*, 7(10), 2006.
- [MGLNS91] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring distributed applications as fragmented objects. Rapport de recherche 1404, INRIA, Rocquencourt, France, March 1991.
- [RKCD01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *NGC2001*, UCL, London, UK, November 2001.
- [Rog98] Dale Rogerson. *Inside COM*. Microsoft Press, 1998.
- [SHT99] M. van Steen, P. Homburg, and A.S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January-March 1999.
- [Sun04] Sun Microsystems, Mountain View, CA, USA. *Java Message Service, Version 1.1*, April 2004.
- [TIB05] TIBCO Software Inc., Palo Alto, CA, USA. *TIB/Rendezvous Concepts, Release 7.4*, 2005.
- [Vin97] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications*, 35(2), February 1997.

- [Vin98] Steve Vinoski. New features for CORBA 3.0. *Communications of the ACM*, 41(10), October 1998.
- [WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the java system. In *Usenix Conference on Object Oriented Technologies and Systems*, March 1996.
- [WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., 1994. http://research.sun.com/techrep/1994/sml_i_tr-94-29.pdf.