Distributed Software Architectures Using Middleware

Introduction

In this essay I will give a brief overview of distributed systems and middleware. The main body of this work will be a discussion of four different families of middleware. Finally, I will give an overview of an emerging middleware system known as publish-subscribe.

Distributed systems

A distributed system is essentially a computer system where components of the system are held on physically seperated, autonomous computers. These machines communicate through the use of a computer network, either a fixed or, in the case of mobile applications, a wireless network. The distributed systems appear to users as a single, integrated computing facility.

In recent years, distributed systems have become increasingly popular and important in modern computing. They provide opportunities for increasing the reliability, availability and performance of applications. However, perhaps the most important feature of a distributed system is that it allows the integration of existing systems. Companies do not wish to rewrite large numbers of legacy applications and a distributed system allows these applications to be integrated in a relatively straightforward manner. A distributed system may comprise components written in a number of different programming languages, running on different operating systems on a variety of computer architectures. In many cases, a distributed system may be cheaper than a single, centralised system. A large number of small, low-power systems may prove cheaper to purchase than a single mainframe or supercomputer. This is the approach employed in Beowulf clusters, which allow a collection of computers to act as a single large computer.

There are obviously many significant disadvantages to distributed systems. They are much more complicated to design, build and maintain than an equivalent centralised system. There are a large number of possible failures that could occur in a distributed system, far more than would be found in a centralised system. Because of this, a distributed system will have multiple points of failure, increasing the likelihood of the system not functioning correctly. Communication over a network will always be far slower and less reliable than communication over a local bus, which has a significant effect on the performance of a distributed system.

Middleware

Middleware is a piece of software lying between the operating system and the application. Middleware, essentially, is a tool to make a distributed system developers life easier. It achieves this by hiding a number of the difficulties involved in building such systems. Since distributed systems may consist of components written in different languages and running on different operating systems, it is often helpful to have a single common development and runtime environment. This is one of the most significant advantages of middleware systems and will make the development of distributed systems easier. Many failures can be dealt with automatically by the middleware without any work needing to be done by the programmer or application. Many types of middleware can provide access transparency, making a remote operation look similar to a local one. Location transparency is another important feature provided by middleware as it means that components can be migrated between computers without any changes required to other components.

Marshalling and unmarshalling, where internal data structures are converted to a format suitable for communication over the network and rebuild at the other end, is a very important consideration in many distributed systems. It can also be a very time consuming task and may have a significant effect on the performance of the final system. Many middleware packages can automate the process, thereby saving programmers a large amount of time.

Obviously, middleware brings its own set of problems to the development of distributed systems. The use of middleware means that more software has to be purchased, installed, tested, maintained and learnt by developers. In many situations, middleware may be unnecessary or even undesirable. Real time applications often require strict guarantees regarding the bandwidth used and time taken by communications that many middleware systems are unable to provide. Also, the marshalling code generated by middleware systems may not be as efficient as code written by the programmer.

There are four main types of middleware that I will discuss in this essay. They are transactional, message-oriented, procedural and object/component middleware.

Transactional middleware

As the name implies, transactional middleware supports the development of systems involving transactions running across multiple hosts. A transaction ensures that the operations required will occur either on all hosts in the system, or no hosts in the system. Transactions are often vital when components on different hosts must be kept in consistent states. Some examples of transactional middleware are BEA’s Tuxedo and Transarc’s Encina.

Transactional middleware uses the two-phase commit (2PC) protocol to implement these transactions. The Distributed Transaction Processing (DTP) protocol defines a programmatic interface for 2PC, which is used by most relational database management systems. This allows servers and database management systems to be easily integrated. Transactional middleware supports both synchronous and asynchronous communication between hosts.

Unfortunately, transactional middleware suffers from a number of disadvantages. Transactions have a significant overhead to manage and the guarantees they provide are often unnecessary or undesirable. If a client is performing long-lived activities, then transactions could prevent other clients from being able to continue. Most transactional middleware systems do not provide any automated marshalling or unmarshalling.

Message-oriented middleware

Message-oriented middleware is a family of middleware that facilitates communication by message exchange. Messages are small pieces of information sent between messages. They can be used to provide a number of functions including event notification and requests for service execution. Message-oriented middleware is particularly well suited for distributed event notification and publish-subscribe systems. Examples of this type of middleware include IBM's MQSeries and Sun's Java Message Queue.

Message-oriented middleware provides an asynchronous service using message queues, but offers no natural support for synchronous communication. Although this may limit its usefulness, it does allow for a number of additional features to be included easily, including fault tolerance, priority schemes and client-server decoupling. This client-server decoupling means that systems built using message-oriented middleware can be very scalable. A further advantage of message-oriented middleware is that it can provide group communication in a transparent manner, although messages will not be delivered in an atomic way to all or no receivers.

Sadly, there are a number of disadvantages to message-oriented middleware. Perhaps the most significant of these is the lack of access transparency. While message queues are a natural approach to communication with remote computers, they are not a natural or efficient method for communication between local components. The lack of access transparency also means that migration and replication transparency are lost. Also, marshalling code has to be written by the programmer, which does make the use of message-oriented middleware harder.

Procedural middleware

Procedural middleware is generally used to provide Remote Procedure Calls (RPC). These are available on a huge number of different operating systems, including most Unix variants and Microsoft Windows systems. RPCs allow server components to be defined using an Interface Definition Language (IDL). From the IDL, it is possible to compile client and server stubs, which then perform the marshalling and unmarshalling and network communication. RPC has bindings for multiple operating systems and programming languages making it a very simple solution for cross-platform distributed system programming.

The cross-platform nature of RPC gives it a huge advantage over other types of middleware. From the developers' perspective, RPC is a familiar method of programming as remote calls are written in the same manner as a local call. The automatically generated marshalling and unmarshalling code also make the process of distributed system development significantly easier.

Procedural middleware, and RPC in particular, suffers from a number of disadvantages, which has resulted in its limited use in modern distributed systems. There is no direct support for multicast or asynchronous communication. Also, RPCs suffer from very limited scalability. There is no direct support for replication or load balancing, meaning that these aspects have to be dealt with directly by the developer, adding a large amount of complexity to the systems. Procedural middleware is not as fault tolerant as other forms of middleware and many possible faults have to be caught and dealt with in the program.

Object & component middleware

Object middleware is an object-oriented extension of procedural middleware adding many features that have appeared in modern object-oriented programming languages. These extensions include support for inheritance, object references and exceptions. Middleware systems in this category include the OMG's CORBA, Microsoft COM, Java RMI and Enterprise Java Beans.

Object middleware shares many of the same advantages as procedural middleware. Again, marshalling and unmarshalling are performed by automatically generated client and server stubs, relieving the programmer from a very error-prone task. Object middleware also has synchronous requests as its default communication mechanism, however many systems include support for asynchronous communications as well. Most object middleware systems also support transactions and messaging, meaning that, in many respects, it can replace all the other types of middleware. This combination of features provides a very powerful and flexible middleware system.

One of the major disadvantages of procedural middleware is its lack of scalability. In many object middleware systems, this is still not fully resolved. Enterprise Java Beans, however, do include support for replication and, therefore, are becoming an increasingly popular solution for distributed software development. Object middleware may not always be applicable in non object-oriented environments and programming languages, which may result in more complicated applications.

Publish-subscribe systems

Publish-subscribe systems are a type of message-oriented middleware that implement a content-based network. In a content-based network, message destinations are decided based on the content of that message, rather than a specific host name or address.

In a conventional message-oriented middleware system, messages are published by a publisher to a specific queue. Clients then retrieve messages from the queues they are interested in. In a publish-subscribe system, clients specify what messages they want to receive (i.e. the content they are interested in). Publishers then send their messages to the network, which will then route them to interested clients. The most obvious advantage of publish-subscribe systems is that clients know that when they receive a message, it is guaranteed to contain information of interest to them. They do not need to waste resources dealing with information they are not interested in.

A subscription is the mechanism used by a client (*subscriber* in publish-subscribe terms) to specify the content it is interested in. These are a list of restrictions on the information that a subscriber wishes to receive and define a subset of messages. The subscription is sent to the subscriber's entry point into the publish-subscribe network, which then forwards it to other nodes in the network. These nodes are commonly referred to as dispatchers. The dispatchers are then responsible for ensuring that the subscriber only receives messages from the subset it has defined.

A publication is the type of message sent by a server (*publisher*) containing information that might be of interest to subscribers. As with subscriptions, they are sent to an entry point into the publish-subscribe network. Dispatchers will then be responsible for ensuring that each message reaches all interested subscribers.

Dispatchers are similar to routers in IP networks, although the routing of messages is more complicated than the routing of IP packets. A forwarding table in each dispatcher contains the subscriptions received, along with the address of the subscriber (note: a subscriber in the table may be another dispatcher). When a dispatcher receives a publication, it is compared against all subscriptions in the forwarding table. If it is in the subset defined by a subscription, then it will be forwarded to the relevant subscriber.

One of the most interesting points about a publish-subscribe system is the anonymity it can provide to both publishers and subscribers. Publishers have no information regarding which, if any, subscribers have received their publications. No dispatchers, apart from the subscriber's entry point, know exactly which subscriber sent a subscription. This, unfortunately, means that payment mechanisms for such systems become much harder. It is possible for publishers to be anonymous, providing many opportunities for the preservation of free speech. It is unlikely, however, that publisher anonymity would be used in practice, as subscribers would probably want to know that the information they receive is from a trustworthy source.

Summary

Distributed systems are an increasingly important field in computing, both in research-oriented environments and in many large companies. Unfortunately, they are complex to build and maintain and can be error prone. Middleware aims to reduce the complexity of such systems by hiding unnecessary details. As with most types of software, there are many different types of middleware, each having different aims and their own set of advantages and disadvantages. There are no good or bad types of middleware; the best choice depends on both the task at hand and the skills of the team who will be using it.