Distributed SW architectures using Middleware

Middleware is once again becoming a hot talking point amongst many circles discussing cutting edge computer technologies, especially as mobile devices become more and more commonplace. However many people do not fully understand what middleware is or have a distorted view of what it is and what can or cannot be achieved by its implementation. The confusion is further added to by the distinct lack of standards in fields such as mobile middleware. Quite simply middleware is connectivity software which allows processes on one device to communicate with processes on another device seamlessly, regardless of operating system or other heterogeneous variables. The middleware software takes care of all the network and OS protocols and provides a simple interface through which processes can communicate over networks. The network could be anything, the Internet, a LAN, WiFi or even mobile phones. Middleware is a huge and growing business with sales in the tens of billions of dollars field. When we have software which is distributed over many clients with multiple servers middleware can be an ideal communication tool. For example Delta Airlines uses middleware to communicate between 40,000 terminals worldwide and their mainframes.

As global integration increases in the real world so it does in the computer world. There is a need to talk to processes which may be running on the other side of the world. As more and more integration happens due to things like expansion and mergers there is a need for better and more flexible middleware architectures, the demand is growing at a phenomenal scale. Integrating computer systems is no easy task, especially when we take into account factors such as bandwidth loads, different workstation architectures, multiple OS and multiple applications. For the programming team to overcome these issues they would need to implement their own middleware which significantly adds to the cost and more importantly time until deployment. So called 'off the shelf' middleware solutions do indeed give an attractive alternative to having to write the communication protocols in house and all the problems that brings. This means that distributed services can be implemented a lot faster and with a lot more ease if some sort of middleware is available. However the benefits and disadvantages of using middleware do need to be examined more closely before informed decisions can be made.

On the benefits side middleware can bring advantages. If the system is distributed there are multiple parts to the system so if one should fail the other parts of the system could carry on the workload. For example we could have multiple component copies over different hosts. If one host were to fail the client could get the component from one of the other hosts, simply 'jumping' hosts till a working component is found. This reduces the impact of host failures which other systems are prone to. Most middleware architectures are also very scalable so there is no need to reinvest in middleware software as the system grows. If there is peak demand at certain times using distributed architectures means the load can be spread to other servers which are relatively load free. This creates a kind of virtual processing power bank which can be very beneficial at keeping hardware costs down. For example one of the city banks now uses middleware to implement their grid computing architecture so that when their is peak demand for processing power they can send work to their computers in Japan or the US depending on where the workload is lowest (dependant on the market opening times in those time zones). Another major benefit is that application development is made simpler; the programmer can concentrate on writing the application, being unencumbered with having to worry about communication protocols. There is no need to look at network architectures, OS incompatibilities etc, the middleware software takes care of it all. When working with multi-tier architectures client-servers can communicate seamlessly. The developer is insulated from all the inner workings and he/she can rely on simple function calls to get a server across the world to do some work.

However one would be forgiven for thinking that middleware is the magic wand solution for all communication problems. The reality is that there is a massive jungle of vendors, all offering different products and hardly any of them compatible with their competitors. This means that the team will be reliant on their vendor for updates and maintenance tying them into a potentially expensive future. It also leads to problems when migrating to new middleware

architectures as some of the code of the applications may need to be recompiled to take account of any new interface the new architecture may have. This is before we even mention the disadvantages of cost, some of the price tags along with middleware architectures are prohibitively expensive. A further disadvantage is the distributed nature of the system. While this is an advantage when using multiple components there is a danger that there are multiple points of failure. The development teams needs to take care that the software is written in such a way that there is no single weakness in the system, such that when one server goes down it does not bring the whole system to its knees.

The current state of middleware is that there are many different architectures available; the choice depends upon the needs of the project. The architectures available range from RPC and MOM to the newer (still academic) Q-CAD and SATIN architectures. Remote Procedure Calls (RPC) is one of the oldest architectures on the block. It is debateable as to whether it is really middleware as there is no separate program but rather RPC is built into the applications which are communicating. The applications on either side of a network invoke stubs which are built into themselves when and if they need to communicate. The stubs are pre-compliled and are part of the application. RPC is arguably the simplest architecture but is now becoming a dated technology as it does not scale well and is not suited to multi-tier architectures. Most RPC implementations do not support peer-2-peer or asynchronous client-server interaction hence is not suited to use with object oriented programs. As the programming world is taking on object-oriented techniques at unprecedented rates RPC is looking like a technology destined for the scrap heap.

Message oriented middleware (MOM) is targeted towards event driven applications. It is a client-server architecture allowing the application to be distributed over many different platforms hence taking the shape of truly independent middleware architecture. MOM makes it easier to create applications that span multiple operating systems and network protocols. MOM takes all responsibility for communication between applications and all the application has to do is send a message to the architecture telling it what to do, the MOM middleware then does all the communication with the server/client. It is also good for object oriented programming as it gives API's that can allow objects to communicate via the middleware seamlessly; developers just need to write to the MOM API. MOM is a generally good architecture but it does have one major drawback, MOM is asynchronous and does not look at network loads. This means that a server could be overloaded by data and the client would keep sending more data regardless of the fact that the server is not reading in data as fast as the client is sending it.

Object Request Broker (ORB) is a middleware solution which as the name suggests is suited particularly well to object oriented technologies. For the developer it encapsulates the details of the network and allows the programmer to build systems by using multiple objects which could be found in multiple locations. This makes it very useful in creating truly distributed software architectures and ORB is the perfect middleware 'glue' to allow the objects to communicate. Other implementations of middleware also exist, such as COM/DCOM, CORBA and JAVA/RMI with their own benefits and disadvantages. However we will now look at the latest middleware technologies which are just emerging.

Mobile devices are growing not only in number but also in features and processing power. At the same time consumers are demanding more and more from their mobile devices. It doesn't take a genius to work out that there is a lot of money to be made in this field. This has led to commercial demand for middleware that allows mobile devices to communicate. This communication is not only with other cellular phones but also for communication with other mobile devices in the environment. These devices could be anything, sensors in the region, servers, network points and all the other associated instruments. It would allow for example people to play mobile games with other people over a network, check the room temperature on your own device and even order a coffee from the waiter! This is where Q-CAD comes to the rescue, an architecture being developed here at UCL. Q-CAD (Quality of service and Context Aware Discovery) allows mobile devices to communicate seamlessly with their environment and access resources. These resources could be anything from services, sensors to downloadable components. Q-CAD automatically handles a request by a mobile device by giving access to the best possible component available at the time. For example if

there are two versions of a game, one designed to run at a higher clock speed than the other it would be pointless sending a low processing power device the higher clock speed application. The Q-CAD system would choose the correct component for the device, in this case the application developed specifically for mobile devices with lower processing capabilities. This allows pervasive computing applications to select the best component available at the time. SATIN middleware is a component model for mobile self organisation. What does this mean? Let's take a look at the background; the problem with many mobile devices is just that, they are mobile! If a user walks out of range of one service as things stand the application cannot use that service. What SATIN proposes is a framework whereby the system could self organise and adapt to the environment it is in and hence give a continuity of service, rather than a fixed and rigid application that cannot work without a networked resource for example. Code mobility and migration allows the system to be a lot more adaptive and hence useful to the end users needs. A change in the environment could thus lead to a change in the application which allows better functionality (a new service perhaps) or allows use with the loss of a service.

As things are virtually all distributed software architectures use some sort of middleware, be it built in or more commonly standalone. The need for better and more flexible middlewares is growing, especially with the growth of mobile computing and the needs associated with wireless networks. There is also demand for standardisation so all developers can produce middleware which works to certain interfaces which makes migration simpler. Middleware needs to be updated to take account of new features such as wireless networks and mobile computers, which is now happening in the academic world. Hopefully the commercial implementation will soon follow.

Word Count: 1874