Distributed Systems

Alex S.[*]

In today's world, we often find ourselves dealing with distributed systems. These take many forms and shapes and are created for all sorts of reasons and purposes. Take the Internet for example, it is a vast network of computers, used mostly for communications. But this is not the limit of a distributed system.

There are several reasons why you might want to consider building or using a distributed system as opposed to a single computer environment.

If you need more CPU processing power than is available in your computer, you might want to offload the CPU work onto other workstations on the network.

If you need more memory or storage capacity, you might offload your data to a machine on the network that has the capacity, or offload it to many nodes across the network.

If you want reliability, you might want to create several clone processes on several machines (in separate places) so that in case one fails, the other ones take over.

There are literally thousands of reasons why you'd want to utilize and work with many computers as opposed to just one. The following discussion will only mention some of the key reasons/techniques, and if you're seriously interested in this area, you should read up just about every book that has 'distributed' in its title.

1 Introduction

Before going on a quest to build a distributed system, you must understand why you need it, and exactly what you will be distributing: memory or CPU, or both. Or do you need it for communications, and/or reliability?

An example of CPU distributed app: If my computer cannot factor a 100 digit number, maybe 1000 computers all across the Internet will be able to do that.

An example of a memory distributed app: I don't have enough memory to hold every song ever recorded, but if I ever need something in particular, perhaps I can find it someplace online.

An example of where you need both CPU and memory: I cannot perform a 1000x1000 matrix multiplication in a reasonable amount of time, so I'll enlist the help of a few thousand computers.

An example of communication: My letters take a few days to arrive via mail, but I can send an e-mail that arrives anywhere in the world in seconds.

[*] `alex@theparticle.com`

An example of reliability: If my company computer goes down, I won't be able to service customers, but if I create a backup computer, then I can just fall back on the one that works in case of emergencies.

Other major reasons include bandwidth - many companies distribute their servers so they can handle huge numbers of users at the same time, with little possibility that any single problem would bring down the whole network.

So as you see, there are many reasons to have distributed systems.

2 Isn't this just Client/Server?

Much of distributed systems bear close resemblenses to the familiar client/server applications that we're used to. On the web, we view a web-server as provider of services, and clients (web-browsers) as clients of services. But the concept of distributed systems goes a bit beyond a single client/server interaction.

In a distributed system, often, the traditional meanings of servers and clients can interact on mostly equal basis. For example, usually, the client has the capability to request things of the server, and the server has the capability to request things of the client. This isn't a requirement, but this call-back feature is often present in distributed system frameworks.

So the answer to the question, isn't this the same as client/server?, is 'yes', and 'no'. It's 'yes', because many of the concepts parallel those in client-server systems, and 'no' because distributed systems go beyond client/server ideas.

3 Frameworks, SDKs, Web-Services?

So far I've mentioned that distributed systems are where we have a few computers, talking to each other. That's about all there is to it.

The complication arises when you try to consider *how* they're talking to each other. What protocols or systems they use to communicate, share resources, cooperate, etc.

3.1 Sockets

Probably one of the lowest level approaches is to design the distributed system based on socket I/O. This level doesn't provide any special distributed system services. It is usually hard (tedious) to program, error prone, etc.

It must be mentioned however, that many large scale systems were written using sockets, and that sockets are the core of just about all the other distributed technologies.

3.2 RPC

One day someone came along and came up with the idea of RPC, or Remote Procedure Call. The idea was to create an interface (and infrastructure) to have a program (often written in C) to call a procedure (function) of another program (also usually written in C) that is running on another computer someplace across the network (or on the same computer).

This happened this way: The call would happen on a proxy function, which would open (if it's not already open) a socket to the remote program, which would accept the call, parameters, etc., perform the function, and send back the results.

This wasn't very standard, and quite complicated. More often than not, it was easier to just open a socket directly and send data, as in plain client/server systems.

3.3 CORBA

Sometime in the early 1990's OMG (Object Management Group) 'invented' a thing called CORBA. It is the Common Object Request Broker Architecture. This is basically a standard Object Oriented RPC mechanism, with servers, and specifications for common services (like transaction management, etc.).

At first, this was mostly for C++, but then people started using it for Java (Iona, Borland).

Because it was standard, it was possible to create client/servers in just about any language. You could have C++ code use a Java object, and vice versa. You could also make use of old COBOL code from Java, etc. It's a fairly flexible infrastructure.

The compatibility was ensured by a tool called IDL, or Interface Definition Language (it is both a tool, and a language). You would create an interface, which is just a something.idl file. You would then run it through an idl2java compiler, to create stubs and skeletons for Java, and you would also run idl2cpp to crate stubs and skeletons in C++.

You could then write Java client code, link in Java stubs, and whenever you made a call on those stub methods, those methods would call the network to the server.

The server can be written in any language; but lets say you wrote it in C++, then you'd have to implement the server skeletons, and link those into the C++ server.

Then the client calls the server the stubs marshals (sends/encodes) the name of the method call, and the parameters to the server's skeletons, which then unmarchals (receives/decodes) the data and calls the server's implementation of the interface.

Both the client and server ran something called an ORB, or Object Request Broker, which would manage all the deployed objects and the communications.

The protocol used by CORBA for this communication is IIOP (or Internet Inter-Orb Protocol). In fact, you could use any protocol (to connect to legacy systems, etc.), but almost everyone just used IIOP.

3.4 RMI

Around the mid 1990s, Java was invented by Sun Microsystems, etc., it became a popular language/platform, everyone started using it, etc. It had great networking support with it's Socket classes, etc. But it was lacking something: ability to use objects remotely.

In the beginning people went around this by either utilizing sockets directly (the most common approach), or spending tons of money for things like OrbixWeb from Iona that provided CORBA for Java (usually big companies with huge projects did this; if you have several hundred possible messages to send over the network, doing it via sockets can get really tedious - so CORBA made for faster development).

Anyway, Java came out with RMI, Remote Method Invocation. Which was basically a Java's version of RPC, only object oriented. It didn't have many fancy features found in CORBA—it was simple and effective.

At the time, you could use it to communicate with Java programs, but then around 1998-1999, Sun Microsystems did something magical, and released RMI/IIOP protocol. That basically meant that you could now use RMI to call CORBA services - which means that you could now use Java to call C++ classes running on another machine, etc., and use most of the goodies (standard services) of CORBA.

This became so popular that people dumped CORBA, and started using Java, and strangely, RMI became the dominant player.

3.5 DCOM

Now, concurrently, through all this CORBA/RMI time, DCOM, a Microsoft's technology to allow COM components to call each other's methods remotely was becoming popular with the Windows programming crowd.

While RMI was standardized on Java, DCOM took a CORBA like approach, and standardized on a binary protocol. The trick is that the binary protocol used Windows services, and components had to be registered in the registry for you to make any use of them, etc., (ie: fairly complicated issues, IMHO).

The gist of it that you could create a component in just about any language, as long as it would run on Windows (you can even do it in Java, but you need Microsoft J++ for that).

It had some major issues (besides for interoperability with other systems). The garbage collection of components was a rather weird design (the client objects would need to ping the server object to prevent it from being cleaned up - this prevents you from having many clients due to network traffic, etc.)

I've only created one DCOM object in my life, so my opinion is very biased against it.

3.6 Messaging

Mostly right from the start of computers, people started sending messages from program to program. There are many infrastructures and concepts about passing messages, the most basic being IP.

These are mostly asynchronous - you send a message and forget it. The infrastructures provide various degree of reliability in delivering that message (many provide *some* reliability, some provide full reliability). Some examples include TIBCO, and MQ Series from IBM (currently known as WebSphereMQ).

These architectures often use the publish/subscribe model, where you subscribe to a particular channel to receive those messages that someone may publish to that channel. There are also concepts like mailboxes, etc., so even if you're not available to pickup the messages right away they're not disappearing.

Many early distributed systems use messaging. Just about all banks/exchanges have some components that deal with TIBCO or MQ Series (like sending around transactions).

3.7 EJB

On the road to make distributed systems easier to implement, EJBs were born. These are mostly components with RMI as the communications mechanism. They're a bit more than just "Objects" though.

EJBs are Enterprise Java Beans. They live in an ejb container, that provides various services; like lifetime management (the container controls when the services are alive, and then they should be cleaned up, etc.)

There are several types of EJBs; Session beans are activated whenever someone calls them, and are very similar to plain RMI objects. A session bean has the capability to maintain its state (it can have variables that stay around as clients call them).

There are also Entity beans; these are mostly representations of the database records. Note that these are still remote, so basically from a remote computer, just by setting some properties on a class, you're in effect changing the database on some other machine.

There are also messaging EJBs; these don't use RMI for communication, but act like interfaces to Messaging infrastructures mentioned above.

Now there are also web-service handling EJBs, but that's better handled in another section.

Basically EJBs tie everything together; databases, logic code, etc., One major limitation (that everyone seems to point out) is that you can't create EJBs in anything other than Java. This is true. However, if you need to use C++ for some component, you can always use CORBA (and RMI/IIOP) to have the EJB talk to the C++ component (or component in any language for that matter).

3.8 Naming Service

It is worth mentioning that as all these distributed systems started to appear, they all had a common thing: the naming service. Distributed objects need to be able to find each other; so just like on the Internet you have DNS, most distributed systems also include some form of a naming directory—to link names to objects. Most of the Windows world uses the registry or directory services, most of the Java world uses JNDI (Java Naming and Directory Interface). CORBA also had a semi-standard[1] naming service running on just about every ORB.

[1]Because of the complexity of CORBA, it was very common for different implementations not to be able to talk to to each other.

Many of these systems can access each other—and pretend like it's perfectly natural for them to do so. For example, JNDI can be used to list your hard-drive directories on your computer, or to access the DNS system, or to query RMI, or CORBA naming services, etc.

Then there's LDAP, Lightweight Directory Access Protocol[2], which aims to provide a standard way (a protocol) of accessing all sorts of directories. This is basically just another fancy interface that seems to be supported by many vendors.

3.9 Web Services

Around year 2000, Microsoft, and just about every IT company on the planet, jumped on the bandwagon of Web Services. These are mostly a replacement for the proprietary messaging systems (MQ Series, etc.), and RPC. They are, however, not a replacement for EJBs, because web-services are stateless components.

The major point is that they're totally cross platform; their protocol basically involves sending around a SOAP envelope. A SOAP envelope is just an XML structure. Because it's XML (plain text), and SOAP is standard, just about anyone on any platform can call and service WebServices. For example, you can create a web-service yourself by using Apache Axis, or simply setting up a server that accepts and responds in XML SOAP messages.

Basically, imagine RPC, but all communications are happening via XML. That's web-services.

Generally, web-services are stateless, and are meant to be used in a heterogeneous environment (Windows desktops talking to UNIX servers :-). They're very well suited towards inter-company communications; the interfaces are automatically published by the WSDL (Web Service Definition Language), which is just an XML file that contains the properties, functions, and method signatures of the web-service. When you connect to the web-service, you can ask for wsdl file, and use that to generate java code (or whatever code) you need to utilize that web-service.

The major drawback that nobody has realized in 2000 is that web-services are inefficient. They waste a lot of bandwidth on sending useless XML tags. It works great for small setups, but if you have many users connecting, those useless xml tags make a really big impact on performance.

There was also little or no provision for security; no standard authentication nor authorization of users. This coupled with the fact that these things are stateless makes web-services a bit less than useful in real life.

It must be mentioned that many folks came up with various extensions—like compressing (via `gzip`) the SOAP messages, or storing it in some binary format, and using lower layer session management (like the popular Cookies that web-servers use). However, all these extensions prevent it from being a 'web-service' (standard, inter-operable, etc.)to begin with.

[2]RFC 1777

3.10 .NET Remoting

As part of the new .NET Framework, and their whole ordeal with Web-Services, Microsoft figured out that the only way to compete in web-services domain is to break the rules. And so they broke them.

.NET Remoting is a technology that's VERY similar to CORBA, except now the components themselves are interpreted instead of being native applications. .NET Remoting also restricts you to working in the Windows world only, but other than that, it's pretty much a rehash of the CORBA idea.

Now, that said, it doesn't mean that's it's horrible. For Microsoft-to-Microsoft communication, it is actually quite flexible. They seem to have learned from the mistakes that CORBA has made (making it over complicated), and made .NET Remoting relatively simple to create, configure, setup, etc., it takes only minutes, and all you need is the .NET Framework to get it going[3]. The distribution of components has been mostly solved (in CORBA, you needed to manually copy files, in .NET you can have them automatically loaded during run time[4]; you can also have them be digitally signed, etc.). A lot of the configuration complication has also been resolved, and the design of filters and sinks is much cleaner and more intuitive.

.NET Remoting is very much like a web-service on steroids. It has the capability to serialize objects in XML SOAP, but not totally compliant with Web-Services (since .NET Remoting components can have states, etc.). Generally however, .NET Remoting is best used via a TCP channel with a binary formatter (ie: same as CORBA, except different protocol). When in this 'native' state, .NET Remoting is quite fast, and is comparable to speeds you'd normally get via low level socket programming—assuming similar communication complexity.

The future is far from certain; where all this is heading. There is a GNU project titled DotGNU, which will probably (with time) support .NET Remoting, and enable programs from other systems to use .NET Services, but I wouldn't count on that. And there's Mono, which is sort of a portable .NET Framework, except its legal status is uncertain because they seem to use parts of Microsoft's framework.

There is also the apparent lag behind J2EE architecture. In J2EE you can do so much with databases, etc., so it's very likely that in future releases Microsoft may somehow tailor its ADO.NET to be more like J2EE's Entity Beans, where modifying a value actually saves changes to database, etc.

4 Other Systems

There are many frameworks, and the above listed just a few primary ones. There are also many protocols and systems that aid in distributed systems, but are not themselves considered distributed systems. Take the synchronization mechanisms, or ways of performing distributed transactions, or synchronizing time, etc.

[3]Note that you don't even need Visual Studio, you just need the *free* .NET Framework SDK and Windows.

[4]You could do that in the Java world since late 1990's.

The above list also failed to mention another important class of software: P2P systems. These are peer-to-peer systems. They're distributed systems where users are relatively equal 'nodes' that form a net, where users can share files/resources. Some P2P architectures include a central server (or a few 'central nodes') others dynamically pick nodes to act as relays, etc.,

There are also many online projects to utilize CPU resources of computers. SETI project uses the resources of thousands of computers (running a SETI screen saver) to look for aliens (abnormal looking signals). A similar 'let us use your CPU' project is Mersenne Prime search, which uses thousands of computers to test primality and look for Mersenne primes.

There are many databases (and web-servers, etc.) that run clustering software, which to a degree can be seen as a distributed system, trying to distribute the load as evenly as possible.

5 Conclusion

Basically, Distributed Systems is a HUGE field, and there are simply too many things to cover in a single lecture. Thus, if you're curious in these topics, I urge everyone to get as many books on distributed systems as possible, and then read them all.