Distributed Computing

Todd Davies

March 30, 2015

Introduction

Many of the most important and visible uses of computer technology rely on distributed computing. Understanding distributed computing requires an understanding of the problems and the challenges stemming from the coordinated operation of different hardware and software. The course focuses on a set of common techniques required to address the key challenges of distributed computing.

Aims

Many of the most important and visible uses of computer technology rely on distributed computing. This course unit aims to build on the course unit in the first year (COMP10052) which introduced students to the principles of distributed computing, and it focuses on techniques and methods in sufficient breadth and depth to provide a foundation for the exploration of specific topics in more advanced course units. The course unit assumes that students have already a solid understanding of the main principles of computing within a single machine, have a rudimentary understanding of the issues related to machine communication and networking, and have been introduced to the area of distributed computing.

- Revision of the characteristics of distributed systems. Challenges. Architectural models.
- Remote Invocation and Distributed Objects
- Java RMI, CORBA, Web Services.
- Message-Oriented middleware
- Synchronous vs asynchronous messaging. Point-to-point messaging. Publish-subscribe.
- Concurrency, co-ordination and distributed transactions
- Ordering of events. Two-phase commit protocol. Consensus.
- Caching and Replication
- Security
- Service-Oriented Architectures, REST and Web Services

Additional reading

Contents

1 Distributed computing

A distributed system is a computing platform build with many computers that:

- Operate concurrently,

- Are physically distributed (and can fail independently)

- Are linked by a network

- Have independent clocks

Leslie Lamport once said that:

> You know you have a distributed system when the crash of a computer youve never heard of stops you from getting any work done. (Leslie Lamport)

The consequences of having a distributed system is that many problems can arise from a lack of synchronization and coordination between parts of the system. Problems include:

- Non-determinism

- Race conditions

- Deadlocks and synchronization

- No notion of a correct time (no global clock)

- No (visible) global state

- Parts of the system may fail independently

Despite these problems, we continue to (and increasingly commonly) build systems and software designed to run on distributed hardware. This is for many reasons, including the fact that people are distributed and move around a lot, information needs to be shared, hardware can be shared to reduce costs or work in parallel etc.

Distributed systems have evolved from simple systems in the 70's and 80's. Early systems were for banks and airline booking systems, but the real proliferation of the technique arose with the internet in the early 90's.

There are eight so called *fallacies* of distributed computing:

1. **The network is reliable**
   The network could stop working at any time for a variety of reasons; hardware failure, malicious actors etc. In order to protect against this, we need to build clever software that can resend failed messages, reorder messages, verify the integrity of messages etc.

2. **Latency is zero**
   Latency is the time it takes from a message to get from one place to another. Even if the data is going at the speed of light, then a packet going from London to the east coast of the USA will take $30 milliseconds$. Developers should make as few calls to networked machines as possible, and transfer as much data as possible each time.

3. **Bandwidth is infinite**
   Bandwidth is how much data you can send in a certain amount of time, and is measured in bits per second. Bandwidth is growing as technology improves, but so do the data requirements of applications, meaning that it is still an issue. Lost packets can reduce bandwidth, so increasing packet size can help. Compression can also be of use.

4. **The network is secure**
   Since networks are largely insecure, you need to think about application security all the time. Implementing access control etc is a good idea for networked applications.

5. **The network topology doesn't change**
   Since we don't control the network, servers could be added or removed, clients can change addresses etc and we won't know in advance. Distributed applications must be adaptive and work around these unexpected changes. The DNS system is a good example.

6. **There is one administrator**
   Different people are in charge of different networks, even different parts of networks. Diagnosing a problem may require the help (and cooperation) of multiple people and organisations.

7. **Transport cost is zero**
   Not only do networks cost money (buying bandwidth, servers etc), but they also cost in terms of computing resources. Serialising between data formats and protocols takes lots of CPU cycles.

8. **The network is homogeneous**
   Interoperability is required for heterogeneous systems to work together properly. Using standard technologies and data formats makes this easier (for example, returning data in JSON format from a REST API instead of a binary blob).

2 Parallelising processes

Many applications can be parallelised by doing homogeneous operations on different processors on different data. If this is the case, in ideal conditions, your speedup will be the same as the number of processors you're using as opposed to using just one processor.

Unfortunately for us, the speedup is not linear, since it takes time to split the data, coordinate the machines and collate the results. There is also a limit to how many processors will keep the speed improving or even keeping constant. If we have more processors than we can actually use, then the overhead of managing them will probably decrease performance, since they'll be doing nothing useful.

It is important to recognise that parallel computing is different to distributed computing. Although they have similar goals and are achieved using similar techniques, parallel computing is usually when you use multiple CPU's in the same computer, whereas distributed computing is using networked computers.

You can still parallelise an application over different systems using the network as a medium. Not all applications will benefit from this; the most suitable applications have CPU intensive sections that don't require much communication between nodes. If the proportion of your application that you can speed up is x (where $0 \leq x \leq 1$), then the maximum speedup you can achieve is $\frac{1}{x}$.

The running time of a program executing on n CPU's, when it runs in t seconds on one cpu is:

$$\text{Running time} = overhead + t\left(1 - x + \frac{x}{p}\right)$$

Where *overhead* is the time it takes to setup, synchronise and communicate between CPU's. In practice, the *overhead* is a function that takes the number of processors as an argument (since it will usually increase as the number of processors increases).

2.1 Finding parallelisable portions of a program

Instructions are well suited to parallel execution if they are either independent of instructions around them (so the result of an instruction doesn't change the result of another), or the same instructions are executed on multiple data (such as mapping over an array).

Loops are a very good source of parallelism, since they are usually responsible for repetitive operations on large amounts of data.

3   Architectures of distributed systems

Broadly, there are two main architectures that can be used in the design of distributed systems; tightly coupled and loosely coupled architectures. Tightly coupled architectures look and (try to) behave as though they were a single computer, whereas loosely coupled architectures are often far more distributed and include client-server, peer-to-peer strategies.

3.1   Distributed Shared Memory

Distributed Shared Memory is a tightly coupled architecture that provides the programmer with an illusion of a single shared memory space. Since the programmer is not concerned with anything lower than the system calls to read and write to memory, and these system calls handle the distributed bit, the programmer can by abstracted from any concerns about message passing.

However, the machines are still connected by a network, and therefore there is a latency between the memory reads and writes and their completion. The middleware will try and minimise network traffic, but depending on the application, this could become a limiting factor.

There are also the issues of keeping track of the physical location of each virtual memory address, since they will be on multiple machines, and replicating the data when needed.

3.2   Loosely coupled architectures

Layered
> In the layered style, messages must flow through a certain number of layers to reach the destination.

Object based
> Here, objects can call other objects.

Event based
> Often used as a publish-subscribe (pub-sub) architecture.
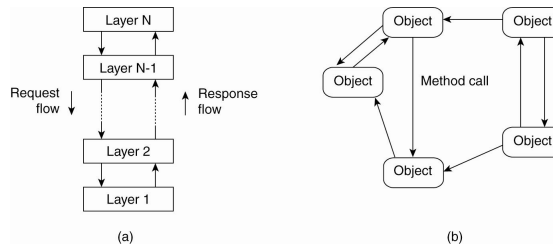
Shared dataspace
> . . .

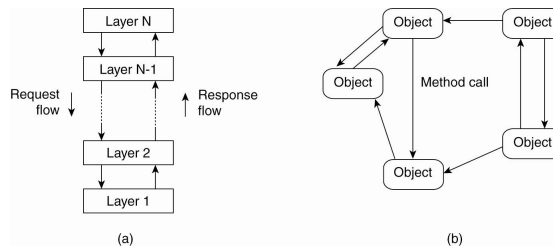Figure 1: The layered architectural style

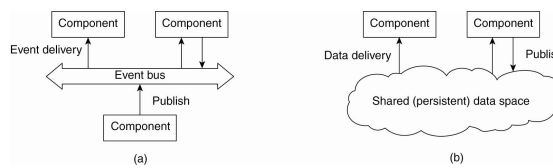Figure 2: The object-based architectural style
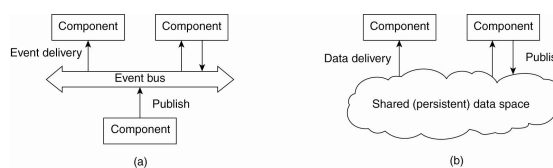

Figure 3: The event-based architectural style

Figure 4: The shared-dataspace architectural style