The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Week 1 (15s1)

Ihor Kuz, Manuel M. T. Chakravarty & Gernot Heiser

Distributed Systems

What is a *distributed system?* Andrew Tannenbaum [TvS07] defines it as

> *A distributed system is a collection of independent computers that appear to its users as a single coherent system.*

This certainly is the ideal form of a distributed system, where the "implementation detail" of building a powerful system out of many simpler systems is entirely hidden from the user.

Are there any such systems? Unfortunately, when we look at the reality of networked computers, we find that the multiplicity of system components usually shines through the abstractions provided by the operating system and other software. In other words, when we work with a collection of independent computers, we are almost always made painfully aware of this. For example, some applications require us to identify and distinguish the individual computers by name while in others our computer hangs due to an error that occurred on a machine that we have never heard of before.

Throughout this course, we will investigate the various technical challenges that ultimately are the cause for the current lack of "true" distributed systems. Moreover, we will investigate various approaches to solving these challenges and study several systems that provide services that are implemented across a collection of computers, but appear as a single service to the user.

For the purpose of this course, we propose to use the following weaker definition of a distributed system,

> *A distributed system is a collection of independent computers that are used jointly to perform a single task or to provide a single service.*

A distributed system by Tannenbaum's definition would surely also be one by our definition; however, our definition is more in line with the current state of the art as perceived by today's users of distributed systems and—not surprisingly—it characterises the kind of systems that we will study throughout this course.

Examples of distributed systems

Probably the simplest and most well known example of a distributed system is the collection of Web servers—or more precisely, servers implementing the HTTP protocol—that jointly provide the distributed database of hypertext and multimedia documents that we know as the World-Wide Web. Other examples include the computers of a local network that provide a uniform view of a distributed file system and the collection of computers on the Internet that implement the Domain Name Service (DNS).

A rather sophisticated version of a distributed system is the XT series of parallel computers by Cray (currently XK7). These are high-performance machines consisting of a collection of computing nodes that are linked by a high-speed low-latency network. The operating system, Cray Linux Environment (CLE) (in the pas also called UNICOS/lc), presents users with a standard Linux environment upon login, but transparently schedules login sessions over a number of available login nodes. However, the implementation of parallel computing jobs on the XT generally requires the programmer to explicitly manage a collection of compute nodes within the application code using XT-specific versions of common parallel programming libraries.

Despite the fact that the systems in these examples are all similar (because they fulfill the definition of a distributed system), there are also many differences between them. The World-Wide Web and DNS, for example, both operate on a global scale. The distributed file system, on the other hand, operates on the scale of a LAN, while the Cray supercomputer operates on an even smaller scale making use of a specially designed high speed network to connect all of its nodes.

Why do we use distributed systems?

The alternative to using a distributed system is to have a huge centralised system, such as a mainframe. For many applications there are a number of economic and technical reasons that make distributed systems much more attractive than their centralised counterparts.

Cost. Better price/performance as long as commodity hardware is used for the component computers.

Performance. By using the combined processing and storage capacity of many nodes, performance levels can be reached that are beyond the range of centralised machines.

Scalability. Resources such as processing and storage capacity can be increased incrementally.

Reliability. By having redundant components the impact of hardware and software faults on users can be reduced.

Inherent distribution. Some applications, such as email and the Web (where users are spread out over the whole world), are naturally distributed. This includes cases where users are geographically dispersed as well as when single resources (e.g., printers, data) need to be shared.

However, these advantages are often offset by the following problems encountered during the use and development of distributed systems:

New component: network. Networks are needed to connect independent nodes and are subject to performance limitations. Besides these limitations, networks also constitute new potential points of failure.

Software complexity. As will become clear throughout this course distributed software is more complex and harder to develop than conventional software; hence, it is more expensive to develop and there is a greater chance of introducing errors.

Failure. With many more computers, networks, and other peripherals making up the whole system, there are more elements that can fail. Distributed systems must be built to survive failure of some of their elements, adding even more complexity to the system software.

Security. Because a distributed system consists of multiple components there are more elements that can be compromised and must, therefore, be secured. This makes it easier to compromise distributed systems.

Hardware and Software Architectures

A key characteristic of our definition of distributed systems is that it includes both a hardware aspect (independent computers) and a software aspect (performing a task and providing a service). From a hardware point of view distributed systems are generally implemented on multicomputers. From a software point of view they are generally implemented as distributed operating systems or middleware.

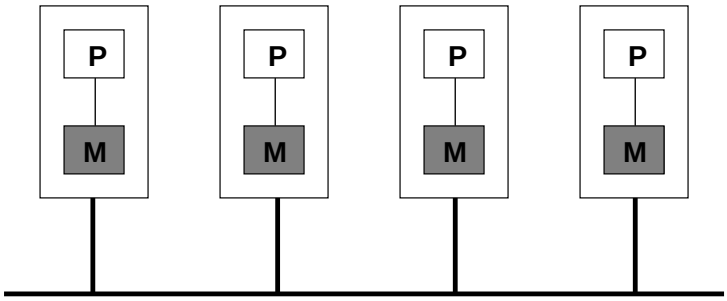

Figure 1: A multicomputer.

Multicomputers

A *multicomputer* consists of separate computing nodes connected to each other over a network (Figure 1). Multicomputers generally differ from each other in three ways:

1. Node resources. This includes the processors, amount of memory, amount of secondary storage, etc. available on each node.

2. Network connection. The network connection between the various nodes can have a large impact on the functionality and applications that such a system can be used for. A multicomputer with a very high bandwidth network is more suitable for applications that actively share data over the nodes and modify large amounts of that shared data. A lower bandwidth network, however, is sufficient for applications where there is less intense sharing of data.

3. Homogeneity. A homogeneous multicomputer is one where all the nodes are the same, that is they are based on the same physical architecture (e.g. processor, system bus, memory, etc.). A heterogeneous multicomputer is one where the nodes are not expected to be the same.

One common characteristic of all types of multicomputers is that the resources on any particular node cannot be directly accessed by any other node. All access to remote resources ultimately takes the form of requests sent over the network to the node where that resource resides.

Distributed Operating System

A *distributed operating system* (DOS) is a an operating system that is built, from the ground up, to provide distributed services. As such, a DOS integrates key distributed services into its architecture (Figure 2). These services may include distributed shared memory, assignment of tasks to processors, masking of failures, distributed storage, interprocess communication, transparent sharing of resources, distributed resource management, etc.

A key property of a distributed operating system is that it strives for a very high level of transparency, ideally providing a single system image. That is, with an ideal DOS users would not be aware that they are, in fact, working on a distributed system.

Distributed operating systems generally assume a homogeneous multicomputer. They are also generally more suited to LAN environments than to wide-area network environments.

In the earlier days of distributed systems research, distributed operating systems where the main topic of interest. Most research focused on ways of integrating distributed services into the operating system, or on ways of distributing traditional operating system services. Currently, however, the emphasis has shifted more toward middleware systems. The main reason for this is that middleware is more flexible (i.e., it does not require that users install and run a particular operating system), and is more suitable for heterogeneous and wide-area multicomputers.

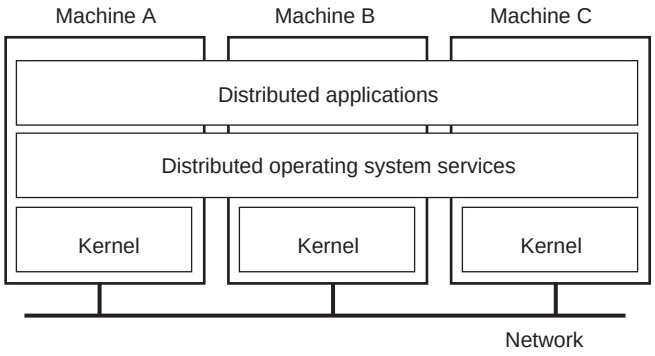

Figure 2: A distributed operating system.

Middleware

Whereas a DOS attempts to create a specific system for distributed applications, the goal of *middleware* is to create system independent interfaces for distributed applications.

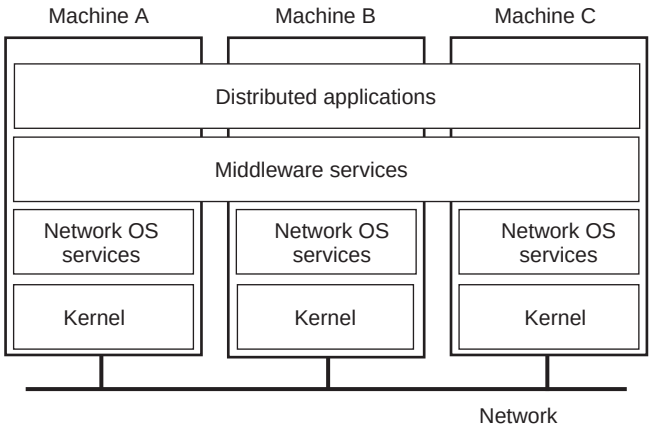

Figure 3: A middleware system.

As shown in Figure 3 middleware consists of a layer of services added between those of a regular *network OS*[1] and the actual applications. These services facilitate the implementation of distributed applications and attempt to hide the heterogeneity (both hardware and software) of the underlying system architectures.

The principle aim of middleware, namely raising the level of abstraction for distributed programming, is achieved in three ways: (1) communication mechanisms that are more convenient and less error prone than basic message passing; (2) independence from OS, network protocol, programming language, etc. and (3) standard services (such as a naming service, transaction service,

[1] A Network OS is a regular OS enhanced with network services such as sockets, remote login, remote file transfer, etc.

security service, etc.).

To make the integration of these various services easier, and to improve transparency and system independence, middleware is usually based on a particular *paradigm*, or model, for describing distribution and communication. Since a paradigm is an overall approach to how a distributed system should be developed, this often manifests itself in a particular programming model such as 'everything is a file', remote procedure call, and distributed objects. 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. Paradigms will be discussed in more detail later on in the course.

Although some forms of middleware focus on adding support for distributed computing directly into a language (e.g., Erlang, Ada, Limbo, etc.), middleware is generally implemented as a set of libraries and tools that enable retrofitting of distributed computing capabilities to existing programming languages. Such systems typically use a central mechanism of the host language (such as the procedure call or method invocation) and dress remote operations up such that they use the same syntax as that mechanism resulting, for example, in remote procedure calls and remote method invocation.

Since an important goal of middleware is to hide the heterogeneity of the underlying systems (and in particular of the services offered by the underlying OS), middleware systems often try to offer a complete set of services so that clients do not have to rely on underlying OS services directly. This provides transparency for programmers writing distributed applications using the given middleware. Unfortunately this 'everything but the kitchen sink' approach often leads to highly bloated systems. As such, current systems exhibit an unhealthy tendency to include more and more functionality in basic middleware and its extensions, which leads to a jungle of bloated interfaces. This problem has been recognised and an important topic of research is investigating adaptive and reflective middleware that can be tailored to provide only what is necessary for particular applications.

With regards to the common paradigms of remote procedure call and remote method invocations, Waldo et al. [WWWK94] have eloquently argued that there is also a danger in confusing local and remote operations and that initial application design already has to take the differences between these two types of operations into account. We shall return to this point later.

Distributed systems and parallel computing

Parallel computing systems aim for improved performance by employing multiple processors to execute a single application. They come in two flavours: shared-memory systems and distributed memory systems. The former use multiple processors that share a single bus and memory subsystem. The latter are distributed systems in the sense of the systems that we are discussing here and use independent computing nodes connected via a network (i.e., a multicomputer). Despite the promise of improved performance, parallel programming remains difficult and if care is not taken performance may end up decreasing rather than increasing.

Distributed systems in context

The study of distributed systems is closely related to two other fields: Networking and Operating Systems. The relationship to networking should be pretty obvious, distributed systems rely on networks to connect the individual computers together. There is a fine and fuzzy line between when one talks about developing networks and developing distributed systems. As we will discuss later the development (and study) of distributed systems concerns itself with the issues that arise when systems are built out of interconnected networked components, rather than the details of communication and networking protocols.

The relationship to operating systems may be less clear. To make a broad generalisation operating systems are responsible for managing the resources of a computer system, and providing access to those resources in an application independent way (and dealing with the issues such as synchronisation, security, etc. that arise). The study of distributed systems can be seen as trying

to provide the same sort of generalised access to distributed resources (and likewise dealing with the issues that arise).

Many distributed applications solve the problems related to distribution in application specific ways. The goal of this course is to examine these problems and provide generalised solutions that can be used in any application. Furthermore we will also examine how these solutions are incorporated into infrastructure software (either distributed OS or middleware) to ease the job of the distributed application developer and help build well functioning distributed applications.

Basic Goals

When considering the design and development of distributed systems, in particular in the context of their distributed nature, there are several key properties that we wish the systems to have. These can be seen as the basic goals of distributed systems.

- Transparency
- Scalability
- Dependability
- Performance
- Flexibility

Providing systems with these properties leads to many of the challenges that we cover in this course.

We discuss the goals in turn.

Transparency

Transparency is the concealment from the user and the application programmer of the separation of the components of a distributed system (i.e., a single image view). Transparency is a strong property that is often difficult to achieve. There are a number of different forms of transparency including the following:

Access Transparency: Local and remote resources are accessed in same way

Location Transparency: Users are unaware of the location of resources

Migration Transparency: Resources can migrate without name change

Replication Transparency: Users are unaware of the existence of multiple copies of resources

Failure Transparency: Users are unaware of the failure of individual components

Concurrency Transparency: Users are unaware of sharing resources with others

Note that complete transparency is not always desirable due to the trade-offs with performance and scalability, as well as the problems that can be caused when confusing local and remote operations. Furthermore complete transparency may not always be possible since nature imposes certain limitations on how fast communication can take place in wide-area networks.

Scalability

Scalability is important in distributed systems, and in particular in wide area distributed systems, and those expected to experience large growth. According to Neuman [Neu94] a system is scalable if:

> *It can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity.*

Adding users and resources causes a system to grow. This growth has three dimensions:

Size: A distributed system can grow with regards to the number of users or resources (e.g., computers) that it supports. As the number of users grows the system may become overloaded (for example, because it must process too many user requests). Likewise as the number of resources managed by the system grows the administration that the system has to perform may become too overwhelming for it.

Geography: A distributed system can grow with regards to geography or the distance between nodes. An increased distance generally results in greater communication delays and the potential for communication failure. Another aspect of geographic scale is the clustering of users in a particular area. While the whole system may have enough resources to handle all users, when they are all concentrated in a single area, the resources available there may not be sufficient to handle the load.

Administration: As a distributed system grows, its various components (users, resources, nodes, networks, etc.) will start to cross administrative domains. This means that the number of organisations or individuals that exert administrative control over the system will grow. In a system that scales poorly with regards to administrative growth this can lead to problems of resource usage, reimbursement, security, etc. In short, an administrative mess.

A claim often made for newly introduced distributed systems (or solutions to specific distributed systems problems) is that they are scalable. These claims of scalability are often (unintentionally) unfounded because they focus on very limited aspects of scalability (for example, a vendor may claim that their system is scalable because it can support up to several hundred servers in a cluster). Although this is a valid claim it says nothing about the scalability with regards to users, or geographic distribution, for example. Another problem with claims of scalability is that many of the techniques used to improve scalability (such as replication), introduce new problems that are often fixed using non-scalable solutions (e.g., the solutions for keeping replicated data consistent may be inherently non-scalable).

Note also that, although a scalable system requires that growth does not affect performance adversely, the mechanisms to make the system scalable may have adverse effects on the overall system performance (e.g., the performance of the system when deployed in a small scale, when scalability is not as important, may be less than optimal due to the overhead of the scalability mechanisms).

The key approach to designing and building a scalable system is *decentralisation*. Generally this requires avoiding any form of centralisation, since this can cause performance bottlenecks. In particular a scalable distributed system must avoid centralising:

- components (e.g., avoid having a single server),
- tables (e.g., avoid having a single centralised directory of names), and
- algorithms (e.g., avoid algorithms based on complete information).

When designing algorithms for distributed systems the following design rules can help avoid centralisation:

- Do not require any machine to hold complete system state.
- Allow nodes to make decisions based on local information.
- Algorithms must survive failure of nodes.
- No assumption of a global clock.

Other, more specific, approaches to avoiding centralisation and improving scalability include: Hiding (or masking) communication delays introduced by wide area networks; Distributing data

over various machines to reduce the load placed on any single machine; and Creating replicas of data and services to reduce the load on any single machine. Besides spreading overall system load out over multiple machines, distribution and replication also help to bring data closer to the users, thus improving geographic scalability. Furthermore, by allowing distribution of data, the management of, and responsibility over, data can be kept within any particular administrative domain, thus helping to improve administrative scalability as well.

Dependability

Although distributed systems provide the potential for higher *availability* due to replication, the distributed nature of services means that more components have to work properly for a single service to function. Hence, there are more potential points of failure and if the system architecture does not take explicit measures to increase reliability, there may actually be a *degradation of availability.* Dependability requires consistency, security, and fault tolerance.

Performance

Any system should strive for maximum performance, but in the case of distributed systems this is a particularly interesting challenge, since it directly conflicts with some other desirable properties. In particular, transparency, security, dependability and scalability can easily be detrimental to performance.

Flexibility

A flexible distributed system can be configured to provide exactly the services that a user or programmer needs. A system with this kind of flexibility generally provides a number of key properties.

Extensibility allows one to add or replace system components in order to extend or modify system functionality.

Openness means that a system provides its services according to standard rules regarding invocation syntax and semantics. Openness allows multiple implementations of standard components to be produced. This provides choice and flexibility.

Interoperability ensures that systems implementing the same standards (and possibly even those that do not) can interoperate.

An important concept with regards to flexibility is the separation of policy and mechanism. A mechanism provides the infrastructure necessary to do something while a policy determines how that something is done. For example, a distributed system may provide secure communication by enabling the encryption of all messages. A system where policy and mechanism is not separated might provide a single hardcoded encryption algorithm that is used to encrypt all outgoing messages. A more flexible system, on the other hand, would provide the infrastructure (i.e., the mechanism) needed to call an arbitrary encryption routine when encrypting outgoing messages. In this way the user or programmer is given an opportunity to choose the most appropriate algorithm to use, rather than a built-in system default.

Component-based architectures are inherently more flexible than monolithic architectures, which makes them particularly attractive for distributed systems.

Common mistakes

Developing distributed systems is different than developing nondistributed ones. Developers with no experience typically make a number of false assumptions when first developing distributed applications [RGO06]. All of these assumptions hold for nondistributed systems, but typically do not hold for distributed systems.

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- The topology does not change
- There is one administrator
- Transport cost is zero
- Everything is homogeneous

Making these assumptions invariably leads to difficulties achieving the desired goals for distributed systems. For example, assuming that the network is reliable will lead to trouble when the network starts dropping packets. Likewise assuming latency is zero will undoubtedly lead to scalability problems as the application grows geographically, or is moved to a different kind of network. The principles that we discuss in the rest of this course deal with the consequence of these assumptions being false, and provide approaches to deal with this.

Principles

There are several key principles underlying all distributed systems. As such, any distributed system can be described based on how those principles apply to that system.

- System Architecture
- Communication
- Synchronisation
- Replication and Consistency
- Fault Tolerance
- Security
- Naming

During the rest of the course we will examine each of these principles in detail.

Paradigms

As mentioned earlier, most middleware systems (and, therefore, most distributed systems) are based on a particular paradigm, or model, for describing distribution and communication. Some of these paradigms are:

- Shared memory
- Distributed objects
- Distributed file system
- Distributed coordination
- Service Oriented Architecture and Web Services
- Distributed databases

- Shared documents
- Agents

Shared memory, distributed objects and distributed file systems, and service oriented architectures will be discussed in detail during this course (time permitting, other paradigms may also be handled, but in less detail). Note, however, that the principles and other issues discussed in the course are relevant to all distributed system paradigms.

Rules of Thumb

Finally, although not directly, or solely, related to distributed systems, we present some rules of thumb that are relevant to the study and design of distributed systems.

Trade-offs: As has been mentioned previously, many of the challenges faced by distributed systems lead to conflicting requirements. For example, we have seen that scalability and overall performance may conflict, likewise flexibility may interfere with reliability. At a certain point trade-offs must be made—a choice must be made about which requirement or property is more important and how far to go when fulfilling that requirement. For example, is it necessary for the reliability requirement to be absolute (i.e., the system must always be available, e.g., even during a natural disaster or war) or is it sufficient to require that the system must remain available only in the face of certain problems, but not others?

Separation of Concerns: When tackling a large, complex, problem (such as designing a distributed system), it is useful to split the problem up into separate concerns and address each concern individually. In distributed systems, this might, for example, mean separating the concerns of communication from those of replication and consistency. This allows the system designer to deal with communication issues without having to worry about the complications of replication and consistency. Likewise, when designing a consistency protocol, the designer does not have to worry about particulars of communication. Approaching the design of a distributed system in this way leads to highly modular or layered systems, which helps to increase a system's flexibility.

End-to-End Argument: In a classic paper Saltzer et al. [SRC84] argue that when building layered systems some functions can only be reliably implemented at the application level. They warn against implementing too much end-to-end functionality (i.e., functionality required at the application level) in the lower layers of a system. This is relevant to the design of distributed systems and services because one is often faced with the question of where to implement a given functionality. Implementing it at the wrong level not only forces everyone to use that, possibly inappropriate, mechanism, but may render it less useful than if it was implemented at a higher (application) level. Implementing encryption as part of the communication layer, for example, may be less secure than end-to-end encryption implemented by the application and therefore offer users a false sense of security.

Policy versus Mechanism: This rule has been discussed previously. Separation of policy and mechanism helps to build flexible and extensible systems, and is, therefore, important to follow when designing distributed systems.

Keep It Simple, Stupid (KISS): Overly complex systems are error prone and difficult to use. If possible, solutions to problems and resulting architectures should be simple rather than mind-numbingly complex.

References

[Neu94] B. Clifford Neuman. Scale in distributed systems. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 463–489. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. `http://clifford.neuman.name/papers/pdf/94--_scale-dist-sys-neuman-readings-dcs.pdf`.

[RGO06] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. `http://www.rgoarchitects.com/Files/fallacies.pdf`, 2006.

[SRC84] J. H. Saltzer, D. P. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), nov 1984. `http://www.reed.com/dpr/docs/Papers/EndtoEnd.html`.

[TvS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, second edition, 2007.

[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/smli_tr-94-29.pdf`.