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

COMP9243 — Lecture 3 (20T3)

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

System Architecture

A distributed system is composed of a number of elements, the most important of which are software components, processing nodes and networks. Some of these elements can be specified as part of a distributed system's design, while others are given (i.e., they have to be accepted as they are). Typically when building a distributed system, the software is under the designer's control. Depending on the scale of the system, the hardware can be specified within the design as well, or already exists and has to be taken as-is. The key, however, is that the software components must be distributed over the hardware components in some way.

The software of distributed systems can become fairly complex—especially in large distributed systems—and its components can spread over many machines. It is important, therefore, to understand how to organise the system. We distinguish between the logical organisation of software components in such a system and their actual physical organisation. The *software architecture* of distributed systems deals with how software components are organised and how they work together, i.e., communicate with each other. Typical software architectures include the layered, object-oriented, data-centred, service-oriented and event-based architectures. Once the software components are instantiated and placed on real machines, we talk about an actual *system architecture*. A few such architectures are discussed in this section. These architectures are distinguished from each other by the roles that the communicating processes take on.

Choosing a good architecture for the design of a distributed system allows splitting of the functionality of the system, thus structuring the application and reducing its complexity. Note that there is no single best architecture—the best architecture for a particular system depends on the application's requirements and the environment.

Client-Server

The client-server architecture is the most common and widely used model for communication between processes. As Figure 1 shows, in this architecture one process takes on the role of a server, while all other processes take on the roles of clients. The server process provides a service (e.g., a time service, a database service, a banking service, etc.) and the clients are customers of that service. A client sends a request to a server, the request is processed at the server and a reply is returned to the client.

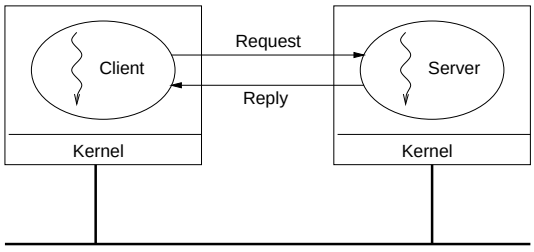

Figure 1: The client-server communication architecture.

A typical client-server application can be decomposed into three logical parts: the interface part, the application logic part, and the data part. Implementations of the client-server architecture vary with regards to how the parts are separated over the client and server roles. A thin

client implementation will provide a minimal user interface layer, and leave everything else to the server. A fat client implementation, on the other hand, will include all of the user interface and application logic in the client, and only rely on the server to store and provide access to data. Implementations in between will split up the interface or application logic parts over the clients and server in different ways.

Vertical Distribution (Multi-Tier)

An extension of the client-server architecture, the vertical distribution, or multi-tier, architecture (see Figure 2) distributes the traditional server functionality over multiple servers. A client request is sent to the first server. During processing of the request this server will request the services of the next server, who will do the same, until the final server is reached. In this way the various servers become clients of each other (see Figure 3). Each server is responsible for a different step (or tier) in the fulfilment of the original client request.

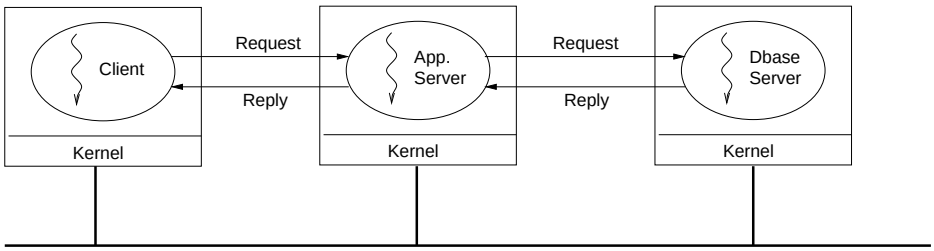

Figure 2: The vertical distribution (multi-tier) communication architecture.

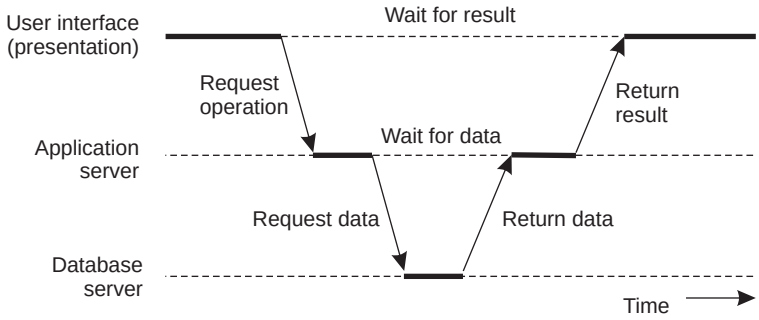

Figure 3: Communication in a multi-tier system.

Splitting up the server functionality in this way is beneficial to a system's scalability as well as its flexibility. Scalability is improved because the processing load on each individual server is reduced, and the whole system can therefore accommodate more users. With regards to flexibility this architecture allows the internal functionality of each server to be modified as long as the interfaces provided remain the same.

Horizontal Distribution

While vertical distribution focuses on splitting up a server's functionality over multiple computers, horizontal distribution involves replicating a server's functionality over multiple computers. A typical example, as shown in Figure 4, is a replicated Web server. In this case each server machine

contains a complete copy of all hosted Web pages and client requests are passed on to the servers in a round robin fashion. The horizontal distribution architecture is generally used to improve scalability (by reducing the load on individual servers) and reliability (by providing redundancy).

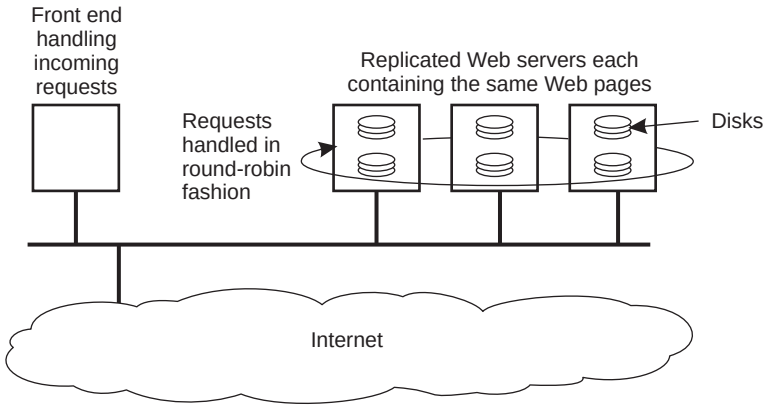

Figure 4: An example of a horizontally distributed Web server.

Note that it is also possible to combine the vertical and horizontal distribution models. For example, each of the servers in the vertical decomposition can be horizontally distributed. Another approach is for each of the replicas in the horizontal distribution model to themselves be vertically distributed.

Peer to Peer

Whereas the previous models have all assumed that different processes take on different roles in the communication architecture, the peer to peer (P2P) architecture takes the opposite approach and assumes that all processes play the same role, and are therefore peers of each other. In Figure 5 each process acts as both a client and a server, both sending out requests and processing incoming requests. Unlike in the vertical distribution architecture, where each server was also a client to another server, in the P2P model all processes provide the same logical services.

Well known examples of the P2P model are file-sharing applications. In these applications users start up a program that they use to search for and download files from other users. At the same time, however, the program also handles search and download requests from other users.

With the potentially huge number of participating nodes in a peer to peer network, it becomes practically impossible for a node to keep track of all other nodes in the system and the information they offer. To reduce the problem, the nodes form an *overlay network*, in which nodes form a virtual network among themselves and only have direct knowledge of a few other nodes. When a node wishes to send a message to an arbitrary other node it must first locate that node by propagating a request along the links in the overlay network. Once the destination node is found, the two nodes can typically communicate directly (although that depends on the underlying network of course).

There are two key types of overlay networks, the distinction being based on how they are built and maintained. In all cases a node in the network will maintain a list of neighbours (called its partial view of the network). In *unstructured* overlays the structure of the network often resembles a random graph. Membership management is typically random, which means that a nodes partial view consists of a random list of other nodes. In order to keep the network connected as nodes join and leave, all nodes periodically exchange their partial views with neighbours, creating a new

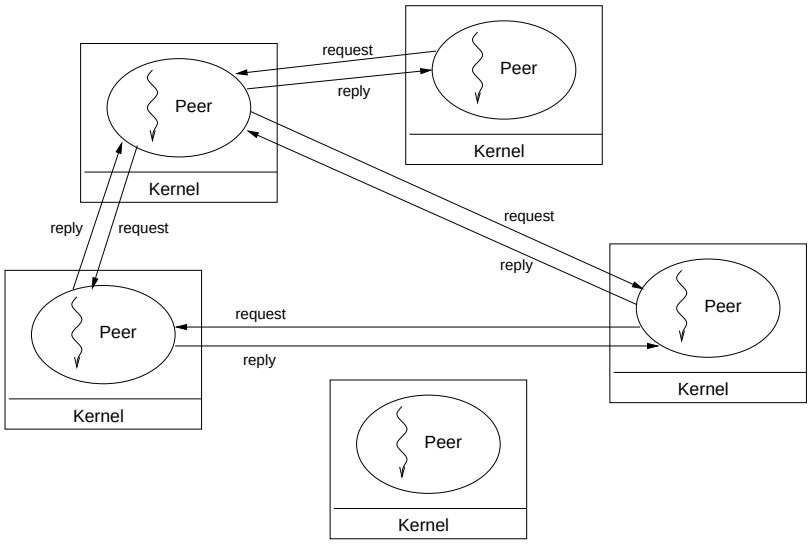

Figure 5: The peer to peer communication architecture.

neighbour list for themselves. As long as nodes both push and pull this information the network tends to stay well connected (i.e., it doesn't become partitioned).

In the case of *structured* overlays the choice of a node's neighbours is determined according to a specific structure. In a distributed hash table, for example, nodes work together to implement a hash table. Each node is responsible for storing the data associated with a range of identifiers. When joining a network, a node is assigned an identifier, locates the node responsible for the range containing that identifier, and takes over part of that identifier space. Each node keeps track of its neighbours in the identifier space. We will discuss specific structured overlays in more detail in a future lecture.

Hybrid

Many more architectures can be designed by combining the previously described architectures in different ways and result in what are called *hybrid architectures*. A few examples are:

Superpeer networks In this architecture a few *superpeers* form a peer to peer network, while the regular peers are clients to a superpeer. This hybrid architecture maintains some of the advantages of a peer to peer system, but simplifies the system by having only the superpeers managing the index of the regular peers, or acting as brokers (e.g., Skype).

Collaborative distributed systems In collaborative distributed systems, peers typically support each other to deliver content in a peer to peer like architecture, while they use a client server architecture for the initial setup of the network. In BitTorrent for example, nodes requesting to download a file from a server first contact the server to get the location of a *tracker*. The tracker then tells the nodes the locations of other nodes, from which chunks of the content can be downloaded concurrently. Nodes must then offer downloaded chunks to other nodes and are registered with the tracker, so that the other nodes can find them.

Edge-server networks In edge-server networks, as the name implies, servers are placed at the "edge" of the Internet, for example at internet service providers (ISPs) or close to enterprise networks. Client nodes (e.g., home users or an enterprise's employees) then access the nearby edge servers instead of the original server (which may be located far away). This architecture is typically well suited for large-scale content-distribution networks such as that provided by Akamai.

Processes and Server Architecture

A key property of all distributed systems is that they consist of separate processes that communicate in order to get work done. Before exploring the various ways that processes on separate computers can communicate, we will first review communication between processes on a single computer (i.e., a uniprocessor or multiprocessor).

Communication takes place between threads of control. There are two models for dealing with threads of control in an operating system. In the *process* model, each thread of control is associated with a single private address space. The threads of control in this model are called *processes*. In the *thread* model, multiple threads of control share a single address space. These threads of control are called *threads*. Sometimes threads are also referred to as lightweight processes because they take up less operating system resources than regular processes.

An important distinction between processes and threads is memory access. Threads share all of their memory, which means that threads can freely access and modify each other's memory. Processes, on the other hand, are prevented from accessing each other's memory. As an exception to this rule, in many systems it is possible for processes to explicitly share memory with other processes.

Some systems provide only a process model, while others provide only a thread model. More common are systems that provide both threads and processes. In this case each process can contain multiple threads, which means that the threads can only freely access the memory of other threads in the same process. In general, when we are not concerned about whether a thread of control is a process or thread, we will refer to it as a process.

A server process in a distributed system typically receives many requests for work from various clients, and it is important to provide quick responses to those clients. In particular, a server should not refuse to do work for one client because it blocked (e.g., because it invoked a blocking system call) while doing work for another client. This is a typical result of implementing a server as a single-threaded process. Alternatives to this are to implement the server using multiple threads, one of which acts as a dispatcher, and the others acting as workers. Another option is to design and build the server as a finite state machine that uses non-blocking system calls.

A key issue in the design of servers is whether they store information about clients or not. In the *stateful* model a server stores persistent information about a client (e.g., which files it has opened). While this leads to good performance since clients do not have to constantly remind servers what their state is, the flipside is that the server must keep track of all its clients (which leads to added work and storage on its part) and must ensure that the state can be recovered after a crash. In the *stateless* model, the server keeps no persistent information about its clients. In this way it does not need to use up resources to track clients, nor does it need to worry about restoring client state after a crash. On the other hand, it requires more communication since clients have to resend their state information with every request.

Often, as discussed above, the server in client-server is not a single machine, but a collection of machines that act as a *clustered server*. In many modern systems separate *virtual machines* are hosted on a single physical machine. This allows consolidation of many servers on a single machine, while providing isolation between them. Since virtual machines can be stopped, migrated, and restarted, virtualisation also provides a good basis for code mobility and load balancing.

Typically the machines (and processes) in such a cluster are assigned dedicated roles including, a logical switch, compute (or application logic) servers, and file or database servers. We have discussed the latter two previously, so now focus on the switch. The role of the switch is to receive client requests and route them to appropriate servers in the cluster. There are several ways to do this. At the lowest level, a transport-layer switch, reroutes TCP connections to other servers. The decision regarding which server to route the request to typically depends on system load. On a slightly higher level, an application switch analyses the incoming request and routes it according to application-specific logic. For example, HTTP requests for HTML files could be routed to one set of servers, while requests for images could be served by other servers. Finally, at an even higher level, A DNS server could act as a switch by returning different IP addresses for a single host name. Typically the server will store multiple addresses for a given name and cycle through

them in a round-robin fashion. The disadvantage of this approach is that the DNS server does not use any application or cluster specific knowledge to route requests.

A final issue with regards to processes is that of code mobility. In some cases it makes sense to change the location where a process is being executed. For example, a process may be moved to an unloaded server in a cluster to improve its performance, or to reduce the load on its current server. Processes that are required to process large amounts of data may be moved to a machine that has the data locally available to prevent the data from having to be sent over the network. We distinguish between two types of code mobility: *weak* mobility and *strong* mobility. In the first case, only code is transfered and the process is restarted from an initial state at its destination. In the second case, the code and an execution context are transfered, and the process resumes execution from where it left off before being moved.