BOOK SUMMARY FOR DISTRIBUTED SYSTEMS Concepts and Design Fifth Edition

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair

Complied By: Nakayiza Hellen (21166042)

Sadullah Karimi (21166041)

CHAPTER 1: CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Distributed systems are everywhere. The Internet enables users throughout the world to access its services wherever they may be located. Each organization manages an intranet, which provides local services and Internet services for local users and generally provides services to other users in the Internet. Small distributed systems can be constructed from mobile computers and other small computational devices that are attached to a wireless network.

Resource sharing is the main motivating factor for constructing distributed systems. Resources such as printers, files, web pages or database records are managed by servers of the appropriate type. For example, web servers manage web pages and other web resources. Resources are accessed by clients – for example, the clients of web servers are generally called browsers.

The construction of distributed systems produces many challenges as shown below:

Heterogeneity: They must be constructed from a variety of different networks, operating systems, computer hardware and programming languages. The Internet communication protocols mask the difference in networks, and middleware can deal with the other differences.

Openness: Distributed systems should be extensible – the first step is to publish the interfaces of the components, but the integration of components written by different Programmers is a real challenge.

Security: Encryption can be used to provide adequate protection of shared resources and to keep sensitive information secret when it is transmitted in messages over a network. Denial of service attacks are still a problem.

Scalability: A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added. The algorithms used to access shared data should avoid performance bottlenecks and data should be structured hierarchically to get the best access times. Frequently accessed data can be replicated.

Failure handling: Any process, computer or network may fail independently of the others. Therefore each component needs to be aware of the possible ways in which the components it depends on may fail and be designed to deal with each of those failures appropriately.

Concurrency: The presence of multiple users in a distributed system is a source of concurrent requests to its resources. Each resource must be designed to be safe in a concurrent environment.

Transparency: The aim is to make certain aspects of distribution invisible to the application programmer so that they need only be concerned with the design of their particular application. For example, they need not be concerned with its location or the details of how its operations are accessed by other components, or whether it will be replicated or migrated. Even failures of networks and processes can be presented to application programmers in the form of exceptions – but they must be handled.

Quality of service: It is not sufficient to provide access to services in distributed systems. In particular, it is also important to provide guarantees regarding the qualities associated with such service access. Examples of such qualities include parameters related to performance, security and reliability.

CHAPTER 2: SYSTEM MODELS

Distributed systems are increasingly complex in terms of their underlying physical characteristics; for example, in terms of the scale of systems, the level of heterogeneity inherent in such systems and the real demands to provide end to end solutions in terms of properties such as security. This places increasing importance on being able to understand and reason about distributed systems in terms of models. Distributed systems can be described in terms of an encompassing architectural model that makes sense of this design space examining the core issues of what is communicating and how these entities communicate, supplemented by consideration of the roles each element may play together with the appropriate placement strategies given the physical distributed infrastructure. Architectural patterns enable more complex designs to be constructed from the underlying core elements, such as the client-server model and major styles of supportive middleware solutions include; solutions based on distributed objects, components, web services and distributed events.

In terms of architectural models, the client-server approach is prevalent – the Web and other Internet services such as FTP, news and mail as well as web services and the DNS are based on this model, as are filing and other local services. Services such as the DNS that have large numbers of users and manage a great deal of information are based on multiple servers and use data partition and replication to enhance availability and fault tolerance. Caching by clients and proxy servers is widely used to enhance the performance of a service. However, there is now a wide variety of approaches to modelling distributed systems including alternative philosophies such as peer-to-peer components or services.

The architectural model is complemented by fundamental models, which aid in reasoning about properties of the distributed system in terms of, for example, performance, reliability and security.

In particular, we presented models of interaction, failure and security. They identify the common characteristics of the basic components from which distributed systems are constructed.

The interaction model is concerned with the performance of processes and communication channels and the absence of a global clock. It identifies a synchronous system as one in which known bounds may be placed on process execution time, message delivery time and clock drift. It identifies an asynchronous system as one in which no bounds may be placed on process execution time, message delivery time and clock drift – which is a description of the behaviour of the Internet.

The failure model classifies the failures of processes and basic communication channels in a distributed system. Masking is a technique by which a more reliable service is built from a less reliable one by masking some of the failures it exhibits. In particular, a reliable communication service can be built from a basic communication channel by masking its failures. For example, its omission failures may be masked by retransmitting lost messages. Integrity is a property of reliable communication – it requires that a message received be identical to one that was sent and that no message be sent twice. Validity is another property – it requires that any message put in the outgoing buffer be delivered eventually to the incoming message buffer.

The security model identifies the possible threats to processes and communication channels in an open distributed system. Some of those threats relate to integrity: malicious users may tamper with messages or replay them. Others threaten their privacy.

Another security issue is the authentication of the principal (user or server) on whose behalf a message was sent. Secure channels use cryptographic techniques to ensure the integrity and privacy of messages and to authenticate pairs of communicating principals.

CHAPTER 3: NETWORKING AND INTERNETWORKING

This chapter focused on the networking concepts and techniques that are needed as a basis for distributed systems, approaching them from the point of view of a distributed system designer.

The performance, reliability, scalability, mobility and quality of service characteristics of the underlying networks impact the behavior of distributed systems and hence affect their design. Changes in user requirements have resulted in the emergence of wireless networks and of high-performance networks with quality of service guarantees.

Packet networks and layered protocols provide the basis for communication in distributed systems. Local area networks are based on packet broadcasting on a shared medium; Ethernet is the dominant technology. Wide area networks are based on packet switching to route packets to their destinations through a connected network. Routing is a key mechanism and a variety of routing algorithms are used, of which the distance-vector method is the most basic but effective. Congestion control is needed to prevent overflow of buffers at the receiver and at intermediate nodes.

Internetworking techniques enable heterogeneous networks to be integrated. Internetworks are constructed by layering a 'virtual' internetwork protocol over collections of networks linked together by routers. The Internet TCP/IP protocols enable computers in the Internet to communicate with one another in a uniform manner, irrespective of whether they are on the same local area network or in different countries.

The Internet standards include many application-level protocols that are suitable for use in wide area distributed applications. IPv6 has the much larger address space needed for the future evolution of the Internet and provision for new application requirements such as quality of service and security. Mobile users are supported by MobileIP for wide area roaming and by wireless LANs based on IEEE 802 standards for local connectivity.

CHAPTER 4: INTERPROCESS COMMUNICATION

The Internet transmission protocols provide two alternative building blocks from which application protocols may be constructed.

There is an interesting trade-off between the two protocols: UDP provides a simple message-passing facility that suffers from omission failures but carries no built-in performance penalties, on the other hand, in good conditions TCP guarantees message delivery, but at the expense of additional messages and higher latency and storage costs.

There are three alternative styles of marshalling. CORBA and its predecessors choose to marshal data for use by recipients that have prior knowledge of the types of its components. In contrast, when Java serializes data, it includes full information about the types of its contents, allowing the recipient to reconstruct it purely from the content. XML, like Java, includes full type information. Another big difference is that CORBA requires a specification of the types of data items to be marshalled (inIDL) in order to generate the marshalling and unmarshalling methods, whereas Java uses reflection in order to serialize objects and deserialize their serial form. But a variety of different means are used for generating XML, depending on the context. For example, many programming languages, including Java, provide processors for translating between XML and language-level objects.

Multicast messages are used in communication between the members of a group of processes. IP multicast provides a multicast service for both local area networks and the Internet. This form of multicast has the same failure semantics as UDP datagrams, but in spite of suffering from omission failures it is a useful tool for many applications of multicast. Some other applications have stronger requirements – in particular, that multicast delivery should be atomic; that is, it should have all-or-nothing delivery. Further requirements on multicast are related to the ordering of messages, the strongest of which requires that all members of a group receive all of the messages in the same order. Multicast can also be supported by overlay networks in cases where, for example, IP multicast is not supported. More generally, overlay networks offer a service

of virtualization of the network architecture, allowing specialist network services to be created on top of underlying networking infrastructure, (for example, UDP or TCP).

Overlay networks partially address the problems associated with Saltzer's end-to-end argument by allowing the generation of more application-specific network abstractions.

The chapter concluded with a case study of the MPI specification, developed by the high-performance computing community and featuring flexible support for message passing together with additional support for multi-way message passing.

CHAPTER 5: REMOTE INVOCATION

There are three paradigms for distributed programming – request-reply protocols, remote procedure calls and remote method invocation. All of these paradigms provide mechanisms for distributed independent entities (processes, objects, components or services) to communicate directly with one another.

Request-reply protocols provide lightweight and minimal support for client-server computing. Such protocols are often used in environments where overheads of communication must be minimized – for example, in embedded systems. Their more common role is to support either RPC or RMI.

The remote procedure call approach was a significant breakthrough in distributed systems, providing higher-level support for programmers by extending the concept of a procedure call to operate in a networked environment. This provides important levels of transparency in distributed systems. However, due to their different failure and performance characteristics and to the possibility of concurrent access to servers, it is not necessarily a good idea to make remote procedure calls appear to be exactly the same as local calls. Remote procedure calls provide a range of invocation semantics, from *maybe* invocations through to *at-most-once* semantics.

The distributed object model is an extension of the local object model used in object-based programming languages. Encapsulated objects form useful components in a distributed system, since encapsulation makes them entirely responsible for managing their own state, and local invocation of methods can be extended to remote invocation.

Each object in a distributed system has a remote object reference (a globally unique identifier) and a remote interface that specifies which of its operations can be invoked remotely.

Middleware implementations of RMI provide components (including proxies, skeletons and dispatchers) that hide the details of marshalling, message passing and locating remote objects from client and server programmers. These components can be generated by an interface compiler. Java RMI extends local invocation to remote invocation using the same syntax, but remote interfaces must be specified by extending an interface called *Remote* and making each method throw a RemoteException. This ensures that programmers know when they make remote invocations or implement remote objects, enabling them to handle errors or to design objects suitable for concurrent access.

CHAPTER 6: INDIRECT COMMUNICATION

Indirect communication is defined in terms of communication through an intermediary, with a resultant uncoupling between producers and consumers of messages. This leads to interesting properties, particularly in terms of dealing with change and establishing fault-tolerant strategies.

Five styles of indirect communication were considered:

- group communication;
- publish-subscribe systems;
- message queues;
- distributed shared memory;

• tuple spaces.

They are similar in terms of all supporting indirect communication through forms of intermediary including groups, channels or topics, queues, shared memory or tuple spaces. Content-based publish-subscribe systems communicate through the publish-subscribe system as a whole, with subscriptions effectively defining logical channels managed by content-based routing. As well as focusing on the commonalities, it is instructive to consider the key differences between the various approaches. All the techniques considered exhibit space uncoupling in that messages are directed to an intermediary and not to any specific recipient or recipients. The position with respect to time uncoupling is more subtle and dependent on the level of persistency in the paradigm. Message queues, distributed shared memory and tuple spaces all exhibit time uncoupling. The other paradigms may, depending on the implementation. For example, in group communication, it is possible in some implementations for a receiver to join a group at an arbitrary point in time and to be brought up-to-date with respect to previous message exchanges (this is an optional feature in JGroups, for example, selected by constructing an appropriate protocol stack). Many publish-subscribe systems do not support persistency of events and hence are not time-uncoupled, but there are exceptions. JMS, for example, does support persistent events, in keeping with its integration of publish-subscribe and message queues.

The next observation is that the initial three techniques (groups, publish-subscribe and message queues) offer a programming model that emphasizes *communication* (through messages or events), whereas distributed shared memory and tuple spaces offer a more *state-based abstraction*. This is a fundamental difference and one that has significant repercussions in terms of scalability; in general terms, the communication based abstractions have the potential to scale to very large scale systems with appropriate routing infrastructure (although this is not the case for group communication because of the need to maintain group membership). In contrast, the two state-based approaches have limitations with respect to scaling. This stems from the need to maintain consistent views of the shared state, for example between multiple readers and writers of shared memory. The situation with tuple spaces is a bit more subtle given the immutable nature of tuples. The key problem rests with implementing the destructive read operation, *take*, in a large-scale system; it is an interesting observation that without this operation, tuple spaces look very much like publish-subscribe systems (and hence are potentially highly scalable).

Most of the above systems also offer *one-to-many* styles of communication, that is, multicast in terms of the communication-based services and global access to shared values in the state-based abstractions. The exceptions are message queuing, which is fundamentally *point-to-point* (and hence often offered in combination with publish subscribe systems in commercial middleware), tuple spaces, which can be either one-to-many or point-to-point depending on whether receiving processes use the *read* or *take* operations, respectively.

There are also differences in *intent* in the various systems. Group communication is mainly designed to support reliable distributed systems, and hence the emphasis is on providing algorithmic support for reliability and ordering of message delivery.

Interestingly, the algorithms to ensure reliability and ordering (especially the latter) can have a significant negative effect on scalability for similar reasons to maintaining consistent views of shared state. Publish-subscribe systems have largely been targeted at information dissemination (for example, in financial systems) and for Enterprise Application Integration. Finally, the shared memory approaches have generally been applied in parallel and distributed processing, including in the Grid community (although tuple spaces have been used effectively across a variety of application domains). Both publish-subscribe systems and tuple space communication have found favor in mobile and ubiquitous computing due to their support for volatile environments. One other key issue associated with the five schemes is that both content-based publish-subscribe and tuple spaces offer a form of *associative addressing* based on content, allowing pattern matching between subscriptions and events or templates against tuples, respectively. The other approaches do not.

	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes

Figure 1: ***Summary of Indirect Communication styles***

Issues related to quality of service in this analysis were not considered. Many message queue systems do offer intrinsic support for reliability in the form of transactions. More generally, however, quality of service remains a key challenge for indirect communication paradigms. Indeed, space and time uncoupling by their very nature make it difficult to reason about end-to-end properties of the system, such as real-time behavior or security, and hence this is an important area for further research.

CHAPTER 7: OPERATING SYSTEM SUPPORT

The operating system supports the middleware layer in providing invocations upon shared resources. It provides a collection of mechanisms upon which varying resource management policies can be implemented, to meet local requirements and to take advantage of technological improvements. It allows servers to encapsulate and protect resources, while allowing clients to share them concurrently. It also provides the mechanisms necessary for clients to invoke operations upon resources.

A process consists of an execution environment and threads: an execution environment consists of an address space, communication interfaces and other local resources such as semaphores; a thread is an activity abstraction that executes within an execution environment. Address spaces need to be large and sparse in order to support sharing and mapped access to objects such as files. New address spaces may be created with their regions inherited from parent processes. An important technique for copying regions is copy-on-write.

Processes can have multiple threads, which share the execution environment.

Multi-threaded processes allow us to achieve relatively cheap concurrency and to take advantage of multiprocessors for parallelism. They are useful for both clients and servers. Recent threads implementations allow for two-tier scheduling: the kernel provides access to multiple processors, while user-level code handles the details of scheduling policy.

The operating system provides basic message-passing primitives and mechanisms for communication via shared memory. Most kernels include network communication as a basic facility; others provide only local

communication and leave network communication to servers, which may implement a range of communication protocols.

This is a trade-off of performance against flexibility.

It was found the proportion of the total time due to software to be relatively large for a null invocation but to decrease as a proportion of the total as the size of the invocation arguments grows. The chief overheads involved in an invocation that are candidates for optimization are marshalling, data copying, packet initialization, thread scheduling and context switching, as well as the flow control protocol used. Invocation between address spaces within a computer is an important special case, and we described the thread-management and parameter-passing techniques used in lightweight RPC.

There are two main approaches to kernel architecture: monolithic kernels and microkernels. The main difference between them lies in where the line is drawn between resource management by the kernel and resource management performed by dynamically loaded (and usually user-level) servers. **A microkernel** must support at least a notion of process and interprocess communication. It supports operating system emulation subsystems as well as language support and other subsystems, such as those for real-time processing. Virtualization offers an attractive alternative to this style by providing emulation of the hardware and then allowing multiple virtual machines (and hence multiple operating systems) to coexist on the same machine.

CHAPTER 8: DISTRIBUTED OBJECTS AND COMPONENTS

This chapter explained the design of complete middleware solutions based around distributed objects and components. As will now be apparent, they represent a natural evolution in thinking about such programming abstractions. Distributed objects are important in terms of bringing the benefits of encapsulation and data abstraction to distributed systems, and as well as associated tools and techniques from the field of object-oriented design. Distributed objects therefore represent a significant step forward from previous approaches based directly on the client-server model. In applying the distributed object approach, however, a number of significant limitations have emerged and these have been presented and analyzed in this chapter. In summary, it is often too complex in practice to use middleware solutions such as CORBA for sophisticated distributed applications and services, particularly when dealing with advanced properties of such systems such as, for example, dependability (fault tolerance and security).

Component technologies overcome these limitations, through their intrinsic separation of concerns between application logic and distributed systems management. The explicit identification of dependencies also helps in terms of supporting third party composition of distributed systems. This chapter examined the EJB 3.0 specification which has made further steps forward in terms of simplifying distributed systems development through an approach that emphasizes the use of plain old Java objects with the complexities managed declaratively through the use of Java annotations. As we saw, more lightweight technologies such as Fractal and OpenCOM have also been introduced to bring the benefits of component-based programming to the development of middleware platforms themselves, with little added overhead in terms of performance.

Component technologies are important for the development of distributed applications, but like any technology, they have their strengths and weaknesses. For example, the component approach is quite prescriptive and best suited to applications that naturally resemble three-tier architectures. To offer a broader perspective on the range of available middleware platforms, the next two chapters examine alternative approaches based on the adoption of web-based standards (web services) and peer-to-peer systems.

CHAPTER 9: WEB SERVICES

Web services have arisen from the need to provide an infrastructure to support interworking between different organizations. This infrastructure generally uses the widely used HTTP protocol to transport messages between clients and servers over the Internet and is based on the use of URIs to refer to resources.

XML, a textual format, is used for data representation and marshalling.

Two separate influences led to the emergence of web services. One of these was the addition of service interfaces to web servers with a view to allowing the resources on a site to be accessed by client programs other than browsers and using a richer form of interaction. The other was the desire to provide something like RPC over the Internet, based on the existing protocols. The resulting web services provide interfaces with sets of operations that can be called remotely. Like any other form of service, a web service can be the client of another web service, thus allowing a web service to integrate or combine a set of other web services.

SOAP is the communication protocol that is generally used by web services and their clients. It can be used to transmit request messages and their replies between client and server, either by the asynchronous exchange of documents or by a form of request reply protocol based on a pair of asynchronous message exchanges. In both cases, the request and reply message is enclosed in an XML-formatted document called an envelope. The SOAP envelope is generally transmitted over the synchronous HTTP protocol, although other transports can be used.

XML and SOAP processors are available for all of the widely used programming languages and operating systems. This enables web services and their clients to be deployed almost anywhere. This form of interworking is enabled by the facts that web services are not tied to any particular programming language and do not support the distributed object model.

In conventional middleware services, interface definitions provide clients with the details of services. However, in the case of web services, service descriptions are used.

A service description specifies the communication protocol to be used (for example,

SOAP) and the URI of the service, as well as describing its interface. The interface may be described either as a set of operations or as a set of messages to be exchanged between client and server.

XML security was designed to provide the necessary protection for the contents of a document exchanged by members of a group of people, who have different tasks to perform on that document. Different parts of the document will be available to different people, some with the ability to add to or alter the content and others only to read it. To enable complete flexibility in its future use, the security properties are defined within the document itself. This is achieved by means of XML, which is a self-describing format. XML elements are used to specify document parts that are encrypted or signed as well as details of the algorithms used and information to help with finding keys. Web services have been used for a variety of purposes in distributed systems. For example, web services provide a natural implementation of the concept of service oriented architecture, in which their loose coupling enables interoperability in Internet scale applications – including business-to-business (B2B) applications. Their inherent loose coupling also supports the emergence of a mashup approach to web service construction. Web services also underpin the Grid, supporting collaborations between scientists or engineers in organizations in different parts of the world. Their work is very often based on the use of raw data collected by instruments at different sites and then processed locally. The Globus toolkit is an implementation of the architecture that has been used in a variety of data-intensive and computationally intensive applications.

Finally, web services are heavily used in cloud computing. For example Amazon's

AWS is based entirely on web service standards coupled with the REST philosophy of service construction.

CHAPTER 10: PEER-TO-PEER SYSTEMS

Peer-to-peer architectures were first shown to support very large scale data sharing with the Internet-wide use of Napster and its descendants for digital music sharing. The fact that much of their use conflicted with copyright laws doesn't diminish their technical significance, although they did also have technical drawbacks that restricted their deployment to applications in which guarantees of data integrity and availability were unimportant.

Subsequent research resulted in the development of peer-to-peer middleware platforms that deliver requests to data objects wherever they are located in the Internet.

In structured approaches, the objects are addressed using GUIDs, which are pure names containing no IP addressing information. Objects are placed at nodes according to some mapping function that is specific to each middleware system. Delivery is performed by a routing overlay in the middleware that maintains routing tables and forwards requests along a route determined by calculating distance according to the chosen mapping function. In unstructured approaches, nodes form themselves into an ad hoc network and then propagate searches through neighbors to find appropriate resources. Several strategies have been developed to improve the performance of this search function and increase the overall scalability of the system.

The middleware platforms add integrity guarantees based on the use of a secure hash function to generate the GUIDs and availability guarantees based on the replication of objects at several nodes and on fault-tolerant routing algorithms.

The platforms have been deployed in several large-scale pilot applications, refined and evaluated. Recent evaluation results indicate that the technology is ready for deployment in applications involving large numbers of users sharing many data objects.

The benefits of peer-to-peer systems include:

• their ability to exploit unused resources (storage, processing) in the host computers;

• their scalability to support large numbers of clients and hosts with excellent balancing of the loads on network links and host computing resources;

• the self-organizing properties of the middleware platforms which result in support costs that are largely independent of the numbers of clients and hosts deployed.

Weaknesses and subjects of current research include:

• their use for the storage of mutable data is relatively costly compared to a trusted, centralized service

• the promising basis that they provide for client and host anonymity has not yetresulted in strong guarantees of anonymity.

CHAPTER 11: SECURITY

Threats to the security of distributed systems are pervasive. It is essential to protect the communication channels and the interfaces of any system that handles information that could be the subject of attacks. Personal mail, electronic commerce and other financial transactions are all examples of such information. Security protocols are carefully designed to guard against loopholes. The design of secure systems starts from a list of threats and a set of 'worst case' assumptions.

Security mechanisms are based on public-key and secret-key cryptography.

Cryptographic algorithms scramble messages in a manner that cannot be reversed without knowledge of the decryption key. Secret-key cryptography is symmetric – the same key serves for both encryption and decryption. If two parties share a secret key, they can exchange encrypted information without risk of eavesdropping or tampering and with guarantees of authenticity.

Public-key cryptography is asymmetric – separate keys are used for encryption and decryption, and knowledge of one does not reveal the other. One key is made public, enabling anyone to send secure messages to

the holder of the corresponding private key and allowing the holder of the private key to sign messages and certificates. Certificates can act as credentials for the use of protected resources.

Resources are protected by access-control mechanisms. Access-control schemes assign rights to principals (that is, the holders of credentials) to perform operations on distributed objects and collections of objects. Rights may be held in access control lists

(ACLs) associated with collections of objects or they may be held by principals in the form of capabilities – unforgeable keys for access to collections of resources.

Capabilities are convenient for the delegation of access rights but are hard to revoke.

Changes to ACLs take effect immediately, revoking the previous access rights, but they are more complex and costly to manage than capabilities.

Until recently, the DES encryption algorithm was the most widely used symmetric encryption scheme, but its 56-bit keys are no longer safe against brute-force attacks. The triple version of DES gives 112-bit key strength, which is safe, but other modern algorithms (such as IDEA and AES) are much faster and provide greater strength.

RSA is the most widely used asymmetric encryption scheme. For safety against factoring attacks, it should be used with 768-bit keys or greater. Public-key (asymmetric) algorithms are outperformed by secret-key (symmetric) algorithms by several orders of magnitude, so they are generally used only in hybrid protocols such as

TLS, for the establishment of secure channels that use shared keys for subsequent exchanges.

The Needham–Schroeder authentication protocol was the first general-purpose, practical security protocol, and it still provides the basis for many practical systems.

Kerberos is a well-designed scheme for the authentication of users and the protection of services within a single organization. Kerberos is based on Needham–Schroeder and symmetric cryptography. TLS is the security protocol designed for and used widely in electronic commerce. It is a flexible protocol for the establishment and use of secure channels based on both symmetric and asymmetric cryptography. The weaknesses of IEEE 802.11 Wi-Fi security provide an object lesson in the difficulties of security design.

CHAPTER 12: DISTRIBUTED FILE SYSTEM

The key design issues for distributed file systems are:

- the effective use of client caching to achieve performance equal to or better than that of local file systems;
- the maintenance of consistency between multiple cached client copies of files when they are updated;
- recovery after client or server failure;
- high throughput for reading and writing files of all sizes;
- scalability.

Distributed file systems are very heavily employed in organizational computing, and their performance has been the subject of much tuning. NFS has a simple stateless protocol, but it has maintained its early position as the dominant distributed file system technology with the help of some relatively minor enhancements to the protocol, tuned implementations and high-performance hardware support.

AFS demonstrated the feasibility of a relatively simple architecture using server state to reduce the cost of maintaining coherent client caches. AFS outperforms NFS in many situations. Recent advances have employed data striping across multiple disks and log-structured writing to further improve performance and scalability.

Current state-of-the-art distributed file systems are highly scalable, provide good performance across both local and wide-area networks, maintain one copy file update semantics and tolerate and recover from failures.

Future requirements include support for mobile users with disconnected operation, and automatic reintegration and quality of service guarantees to meet the need for the persistent storage and delivery of streams of multimedia and other time-dependent data.

CHAPTER 13: NAME SERVICES

This chapter described the design and implementation of name services in distributed systems. Name services store the attributes of objects in a distributed system in particular, their addresses and return these attributes when a textual name is supplied to be looked up.

The main requirements for the name service are an ability to handle an arbitrary number of names, a long lifetime, high availability, the isolation of faults and the tolerance of mistrust.

The primary design issue is the structure of the name space the syntactic rules governing names. A related issue is the resolution model, which sets out the rules by which a multi-component name is resolved to a set of attributes. The set of bound names must be managed. Most designs consider the name space to be divided into domains discrete sections of the name space, each of which is associated with a single authority controlling the binding of names within it. The implementation of the name service may span different organizations and user communities. The collection of bindings between names and attributes, in other words, is stored at multiple name servers, each of which stores at least part of the set of names within a naming domain. The question of navigation therefore arises by what procedure can a name be resolved when the necessary information is stored at several sites? The types of navigation that are supported are iterative, multicast, recursive server-controlled and non-recursive server-controlled.

Another important aspect of the implementation of a name service is the use of replication and caching. Both of these assist in making the service highly available, and both also reduce the time taken to resolve a name.

There are two main cases of name service design and implementation. The Domain Name System is widely used for naming computers and addressing electronic mail across the Internet; it achieves good response times through replication and caching. The Global Name Service is a design that has tackled the issue of reconfiguring the name space as organizational changes occur.

Directory services provide data about matching objects and services when clients supply attribute-based descriptions. X.500 is a model for directory services that can range in scope from individual organizations to global directories. It has been taken up more widely for use in intranets since the arrival of the LDAP software.

CHAPTER 14: TIME AND GLOBAL STATES

Accurate timekeeping is important for distributed systems. There are different algorithms used for synchronizing clocks despite the drift between them and the variability of message delays between computers.

The degree of synchronization accuracy that is practically obtainable fulfils many requirements but is nonetheless not sufficient to determine the ordering of an arbitrary pair of events occurring at different computers. The happened-before relation is a partial order on events that reflects a flow of information between them within a process, or via messages between processes. Some algorithms require events to be ordered in happened-before order, for example, successive updates made to separate copies of data. Lamport clocks are counters that are updated in accordance with the happened-before relationship between events. Vector clocks are an improvement on Lamport clocks, in that it is possible to determine by examining their vector timestamps whether two events are ordered by happened-before or are concurrent.

We introduced the concepts of events, local and global histories, cuts, local and global states, runs, consistent states, linearizations (consistent runs) and reachability. A consistent state or run is one that is in accord with the happened-before relation.

There is a problem with recording a consistent global state by observing a system's execution. Our objective was to evaluate a predicate on this state.

An important class of predicates are the stable predicates. The snapshot algorithm of Chandy and Lamport, captures a consistent global state and allows us to make assertions about whether a stable predicate holds in the actual execution. The authors gave Marzullo and Neiger's algorithm for deriving assertions about whether a predicate held or may have held in the actual run. This algorithm employs a monitor process to collect states. The monitor examines vector timestamps to extract consistent global states, and it constructs and examines the lattice of all consistent global states.

This algorithm involves great computational complexity but is valuable for understanding and can be of some practical benefit in real systems where relatively few events change the global predicate's value. The algorithm has a more efficient variant in synchronous systems, where clocks may be synchronized.

CHAPTER 15: COORDINATION AND AGREEMENT

Processes are needed to access shared resources under conditions of mutual exclusion. Locks are not always implemented by the servers that manage the shared resources, and a separate distributed mutual exclusion service is then required. Three algorithms were considered that achieve mutual exclusion: one employing a central server, a ring-based algorithm and a multicast-based algorithm using logical clocks. None of these mechanisms can withstand failure as we described them, although they can be modified to tolerate some faults.

Considering a ring-based algorithm and the bully algorithm, the common aim is to elect a process uniquely from a given set even if several elections take place concurrently. The bully algorithm could be used, for example, to elect a new master time server, or a new lock server, when the previous one fails.

For coordination and agreement in group communication, there is reliable multicast, in which the correct processes agree on the set of messages to be delivered, and multicast with FIFO, causal and total delivery ordering.

There are three problems of consensus that is; Byzantine generals and interactive consistency. The authors showed relationships between these problems including the relationship between consensus and reliable, totally ordered multicast.

Solutions exist in a synchronous system, and we described some of them. In fact, solutions exist even when arbitrary failures are possible. We outlined part of the solution to the Byzantine generals problem of Lamport *et al.* [1982]. More recent algorithms have lower complexity, but in principle none can better the $f + 1$ rounds taken by this algorithm, unless messages are digitally signed.

Nonetheless, systems regularly do reach agreement in asynchronous systems.

CHAPTER 16: TRANSACTIONS AND CONCURRENCY CONTROL

Transactions provide a means by which clients can specify sequences of operations that are atomic in the presence of other concurrent transactions and server crashes. The first aspect of atomicity is achieved by running transactions so that their effects are serially equivalent. The effects of committed transactions are recorded in permanent storage so that the transaction service can recover from process crashes. To allow transactions the ability to abort, without having harmful side effects on other transactions, executions must be strict – that is, reads and writes of one transaction must be delayed until other transactions that wrote the same objects have either committed or aborted. To allow transactions the choice of either committing or aborting, their operations are performed in tentative versions that cannot be accessed by other transactions. The tentative versions of objects are copied to the real objects and to permanent storage when a transaction commits.

Nested transactions are formed by structuring transactions from other sub transactions.

Nesting is particularly useful in distributed systems because it allows concurrent execution of sub transactions in separate servers and independent recovery of parts of a transaction.

Operation conflicts form a basis for the derivation of concurrency control protocols. Protocols must not only ensure serializability but also allow for recovery by using strict executions to avoid problems associated with transactions aborting, such as cascading aborts.

Three alternative strategies are possible in scheduling an operation in a transaction. They are (1) to execute it immediately, (2) to delay it or (3) to abort it.

Strict two-phase locking uses the first two strategies, resorting to abortion only in the case of deadlock. It ensures serializability by ordering transactions according to when they access common objects. Its main drawback is that deadlocks can occur.

Timestamp ordering uses all three strategies to ensure serializability by ordering transactions' accesses to objects according to the time transactions start. This method cannot suffer from deadlocks and is advantageous for read-only transactions. However, transactions must be aborted when they arrive too late.

Multiversion timestamp ordering is particularly effective.

Optimistic concurrency control allows transactions to proceed without any form of checking until they are completed. Transactions are validated before being allowed to commit. Backward validation requires the maintenance of multiple write sets of committed transactions, whereas forward validation must validate against active transactions and has the advantage that it allows alternative strategies for resolving conflicts. Starvation can occur due to repeated aborting of a transaction that fails validation in optimistic concurrency control and even in timestamp ordering.

CHAPTER 17: DISTRIBUTED TRANSACTIONS

In the most general case, a client's transaction will request operations on objects in several different servers. **A distributed transaction** is any transaction whose activity involves several different servers. A nested transaction structure may be used to allow additional concurrency and independent committing by the servers in a distributed transaction.

The atomicity property of transactions requires that the servers participating in a distributed transaction either all commit it or all abort it. Atomic commit protocols are designed to achieve this effect, even if servers crash during their execution. **The two-phase commit protocol** allows a server to decide to abort unilaterally. It includes timeout actions to deal with delays due to servers crashing. The two-phase commit protocol can take an unbounded amount of time to complete but is guaranteed to complete eventually.

Concurrency control in distributed transactions is modular – each server is responsible for the serializability of transactions that access its own objects. However, additional protocols are required to ensure that transactions are serializable globally.

Distributed transactions that use timestamp ordering require a means of generating an agreed timestamp ordering between the multiple servers. Those that use optimistic concurrency control require global validation or a means of forcing a global ordering on committing transactions.

Distributed transactions that use two-phase locking can suffer from distributed deadlocks. The aim of distributed deadlock detection is to look for cycles in the global wait-for graph. If a cycle is found, one or more transactions must be aborted to resolve the deadlock. Edge chasing is a non-centralized approach to the detection of distributed deadlocks.

Transaction-based applications have strong requirements for the long life and integrity of the information stored, but they do not usually have requirements for immediate response at all times. Atomic commit protocols are the key to distributed transactions, but they cannot be guaranteed to complete within a particular time limit.

Transactions are made durable by performing checkpoints and logging in a recovery file, which is used for recovery when a server is replaced after a crash. Users of a transaction service will experience some delay during recovery. Although it is assumed that the servers of distributed transactions exhibit crash failures and run in an asynchronous system, they are able to reach consensus about the outcome of transactions because crashed servers are replaced with new processes that can acquire all the relevant information from permanent storage or from other servers.

CHAPTER 18: REPLICATION

Replicating objects is an important means of achieving services with good performance, high availability and fault tolerance in a distributed system. We described architectures for services in which replica managers hold replicas of objects, and in which front ends make this replication transparent. Clients, front ends and replica managers may be separate processes or exist in the same address space.

Each logical object is implemented by a set of physical replicas. Often, updates to these replicas can be made conveniently by group communication.

Linearizability and Sequential Consistency are the correctness criteria for fault-tolerant services. These criteria express how the services must provide the equivalent of a single image of the set of logical objects, even though those objects are replicated. The most practically significant of the criteria is sequential consistency.

In passive (primary-backup) replication, fault tolerance is achieved by directing all requests through a distinguished replica manager and having a backup replica manager take over if this fails. In active replication, all replica managers process all requests independently. Both forms of replication can be conveniently implemented using group communication.

Both Gossip and Bayou allow clients to make updates to local replicas while partitioned. In each system, replica managers exchange updates with one another when they become reconnected. Gossip provides its highest availability at the expense of relaxed, causal consistency. Bayou provides stronger eventual consistency guarantees, employing automatic conflict detection and the technique of operational transformation to resolve conflicts. Coda is a highly available file system that uses version vectors to detect potentially conflicting updates.

Both primary-backup architectures and architectures in which front ends may communicate with any replica manager exist for this case. Transactional systems allow for replica manager failures and network partitions. The techniques of available copies, quorum consensus and virtual partitions enable operations within transactions to make progress even in some circumstances where not all replicas are reachable.

CHAPTER 19: MOBILE AND UBIQUITOUS COMPUTING

Most of the challenges raised by mobile and ubiquitous computer systems stem from the fact that those systems are volatile, which in turn is largely due to the fact that they are integrated with our everyday physical world. The systems are volatile in that the set of users, hardware and software components in a given smart space is subject to unpredictable change. Components tend to make and break associations routinely, either as they move from smart space to smart space or because of failure. Connection bandwidth can vary widely over time. Components can fail as batteries die or for other reasons.

The integration of devices with our physical world involves sensing and context awareness and there are some architectures for processing sensed data. But there remains a challenge that might be described as *physical fidelity*: how accurately can a system with sensing and computation behave in accordance with the subtle semantics we humans associate with the physical world in which we live? Does a 'context aware phone' really behave as we would want in inhibiting rings appropriately as we move between places? Does the web presence of a place such as a hotel room in Cooltown actually record all the web-present entities that a human would say were in that place – and not, for example, some in an adjacent room?

Security and privacy feature strongly in research on mobile and ubiquitous systems. Volatility complicates security because it begs the question of what basis there can be for trust between components that wish to establish a secure channel.

Fortunately, the existence of physically constrained channels goes some way to enabling secure channels where there is a human present. Physical integration has implications for privacy: if the user is being tracked to provide them with context-aware services, then there may be a severe loss of privacy.

Physical integration also means new degrees of constraint in terms of such factors as device energy capacity, wireless bandwidth and user interfaces – a node in a sensor network has little of the first two and none of the last; a mobile phone has more of all three, but still much less than a desktop machine.

The architecture of Cooltown project as a case study is distinctive in that it applies lessons learned from the Web to ubiquitous computing. The advantage is a high degree of interoperability. But as a result, the Cooltown design applies mainly to situations in which humans supervise interactions.

The differences between mobile and ubiquitous systems and the more conventional distributed systems are largely to do with aspects of volatility and physical integration.

CHAPTER 20: DISTRIBUTED MULTIMEDIA SYSTEMS

Multimedia applications require new system mechanisms to enable them to handle large volumes of time-dependent data. The most important of these mechanisms are concerned with quality of service management. They must allocate bandwidth and other resources in a manner that ensures that application resource requirements can be met, and they must schedule the use of the resources so that the many fine-grained deadlines of multimedia applications are met.

Quality of service management handles QoS requests from applications, specifying the bandwidth, latency and loss rates acceptable for multimedia streams, and it performs admission control, determining whether sufficient unreserved resources are available to meet each new request and negotiating with the application if necessary. Once a QoS request is accepted, the resources are reserved and a guarantee is issued to the application.

The processor capacity and network bandwidth allocated to an application must then be scheduled to meet the application's needs. A real-time processor scheduling algorithm such as earliest-deadline-first or rate-monotonic is required to ensure that each stream element is processed in time.

Traffic shaping is the name given to algorithms that buffer real-time data to smooth out the timing irregularities that inevitably arise. Streams can be adapted to utilize fewer resources by reducing the bandwidth of the source (scaling) or at points along the way (filtering).

The Tiger video file server is an excellent example of a scalable system that provides stream delivery on a potentially very large scale with strong quality of service guarantees. Its resource scheduling is highly specialized, and it offers an excellent example of the changed design approach that is often required for such systems. The other two case studies, BitTorrent and ESM, also provide strong examples of how to support the downloading and real-time streaming of video data, respectively, again highlighting the impact that multimedia has on systems design.

CHAPTER 21: DESIGNING DISTRIBUTED SYSTEMS (GOOGLE CASE STUDY)

This chapter concludes the book by addressing the key issue of how one very large

Internet enterprise has approached the design of a distributed system to support demanding set of real-world applications. This is a very challenging topic and one that requires a thorough understanding of the technological choices available to distributed systems developers at all levels of system development, including communication paradigms, available services and associated distributed algorithms. The inevitable trade-offs associated with the design choices demand a thorough understanding of the application domain.

The approach taken in this chapter is to highlight the art of distributed systems design through a substantial case study – that is, the examination of the design of the underlying Google infrastructure, the platform and middleware used by Google to support its search engine and expanding set of applications and services. This is a compelling case study as it addresses what is the most complex and large-scale distributed system ever constructed, and one that has demonstrably met its design requirements.

This case study examined the overall architecture of the system together with indepth studies of the key underlying services – specifically, protocol buffers, the publishsubscribe service, GFS, Chubby, Bigtable, MapReduce and Sawzall – which all work together to support complex distributed applications and services including the core search engine and Google Earth. One key lesson to be taken from this case study is the importance of really understanding your application domain, deriving a core set of underlying design principles and applying them consistently. In the case of Google, this manifests itself in a strong advocacy of simplicity and low-overhead approaches coupled with an emphasis on testing, logging and tracing. The end result is an architecture that is highly scalable, reliable, high performance and open in terms of supporting new applications and services.

The Google infrastructure is one of a number of middleware solutions for cloud computing that have emerged in recent years (albeit only fully available within Google

Other solutions include the Amazon Web Services (AWS) [aws.amazon.com],

Microsoft's Azure [www.microsoft.com IV] and open source solutions including

Hadoop (which includes an implementation of MapReduce) [hadoop.apache.org],

Eucalyptus [open.eucalyptus.com], the Google App Engine (available externally and providing a window on some but not all of the functionality offered by the Google infrastructure) [code.google.com IV] and Sector/Sphere [sector.sourceforge.net].

OpenStreetMap [www.openstreetmap.org], an open alternative to Google Maps that operates in a similar manner using voluntarily developed software and non-commercial servers, has also been developed. Details of these implementations are generally available and the reader is encouraged to study a selection of these architectures, comparing the design choices with those presented in the above case study.

Beyond that, there is a real paucity of published case studies related to distributed systems design, and this is a pity given the potential educational value of studying overall distributed systems architectures and their associated design principles. The main contribution of this chapter was therefore to provide a first in-depth case study illustrating the complexities of designing and implementing a complete distributed system solution.