ECE 454 Course Notes

Distributed Computing

Michael Socha

4A Software Engineering
University of Waterloo
Spring 2018

Contents

1 Course Overview

1.1 Logistics

- **Professor:** Wojciech Golab

1.2 Overview of Topics

This course is intended to provide an introduction into distributed systems. Topics covered include:

- Some of the general architectures, protocols, and algorithms underlying modern distributed systems
- Techniques for making distributed systems scalable and dependable, as well as the relevant trade-offs
- Remote procedure call (RPC) frameworks for implementing distributed services
- Scalable data processing frameworks for solving analytics problems
- Fault-tolerant coordination services for configuration management, synchronization, and failure-detection in distributed software systems
- Stream-processing engines for real-time analytics
- Relative merits of distributed versus centralized systems

2 Introduction

2.1 What is a Distributed System?

A distributed system is a collection of autonomous computing elements that appear to its users to be a single coherent system.

2.2 Rationale Behind Distributed Systems

Distributed systems are often used for the following reasons:

- Resource sharing
- Integrating multiple systems
- Centralized systems may not be as effective as many smaller systems working together
- Users themselves may be mobile and distributed physically from one another

2.3 Middleware

Middleware is a layer of software that separates applications from their underlying platform. Middleware is often used by distributed systems to support heterogeneous computers and networks while offering a single-system view. Common middleware services include:

- Communication (e.g. add job to remote queue)
- Transactions (e.g. access multiple independent services atomically)
- Service composition (e.g. Independent map system enhanced with independent weather forecast system)
- Reliability (e.g. replicated state machine)

2.4 Goals of Distributed Systems

2.4.1 Supporting Resource Sharing

Resources can include:

- Peripheral devices (e.g. printers, video cameras)
- Storage facilities (e.g. file servers)
- Enterprise data (e.g. payroll)
- Web page (e.g. Web search)

- CPUs (e.g. supercomputers)

2.4.2 Making Distribution Transparent

Distribution transparency attempts to hide that processes and resources are physically distributed. Types of transparency include:

- **Access:** Hide differences in data representation and how data is accessed
- **Location:** Hide where resource is located
- **Migration:** Hide that a resource may move to another location
- **Relocation:** Hide that a resource may move to another location while in use
- **Replication:** Hide that a resource is replicated
- **Concurrency:** Hide that a resource may be shared by users competing for that resource
- **Failure:** Hide the failure and recovery of a resource

2.4.3 Being Open

An open distributed system offers components that can be easily used by or integrated into other systems. Key properties of openness include:

- Interoperability
- Composability
- Extensibility
- Separation of policy from mechanism

2.4.4 Being Scalable

Scalability covers a system's ability to expand along three axes:

- Size (e.g. adding users and resources)
- Geography (e.g. users across large distances)
- Administration (e.g. multiple independent admins)

Centralized systems tend to have limited scalability.

A few common scaling techniques include:

- Hiding communication latencies (i.e. trying to avoid waiting for responses from remote-server machines)

- Partitioning (i.e. taking a component, splitting it into smaller parts, and spreading those parts across the system)
- Replication (i.e. adding multiple copies of a component to increase availability, balance load, support caching, etc.)

2.5 Common Fallacies of Distributed Computing

The following are common beginner assumptions that can lead to major trouble:

- The network is reliable
- The network is secure
- The network is homogeneous
- The network topology is static
- Latency is 0
- Bandwidth is unlimited
- Transport cost is 0
- There is one administrator

2.6 Types of Distributed Systems

- **Web sites and Web services**
- **High performance computing (HPC):** High-performance computing in distributed memory settings, where message communication is used instead of shared memory.
- **Cluster Computing:** Distributing CPU or I/O intensive tasks over multiple servers.
- **Cloud and Grid Computing:** Grid computing focuses on combining resources from different institutions, while cloud computing provides access to shared pools of configurable system resources.
- **Transaction Processing:** Distributed transactions coordinated by transaction processing (TP) monitor.
- **Enterprise Application Integration (EAI):** Middleware often used as communication facilitator in such systems.
- **Sensor Networks:** Each sensor in a network can process and store data, which can be queried by some operator. Sensors often rely on in-network data processing to reduce communication costs.

3 Architectures

3.1 Definitions

- **Component:** A modular unit with a well-defined interface
- **Connector:** Mechanism that mediates communication, coordination, or cooperation among components
- **Software architecture:** Organization of software components
- **System architecture:** Instantiation of software architecture in which software components are placed on real machines
- **Autonomic System:** System that adapts to its environment by monitoring its own behavior and reacting accordingly

3.2 Architectural Styles

3.2.1 Layered

Control flows from layer-to-layer; requests flow down the hierarchy and responses flow up, with each layer only interacting with its two neighboring layers. Layered architectures are often used to support client-server interactions, where a client component requests a service from a server component and waits for a response. Many enterprise systems are organized intro three layers, namely a user interface, application layer, and database. Note that middle layers may act as a client to the layer below and a server to the layer above.

Vertical distribution describes the logical layers of a system being organized as separate physical tiers. Horizontal distribution describes a single logical layer being split across multiple machines.

3.2.2 Object-Based Architecture

Components are much more loosely organized than in a layered architecture, with components being able to interact freely with one another and no strict concept of layer.

3.2.3 Data-Centered Architecture

Components communicate by using a shared data repository, such as a database or file system.

3.2.4 Event-Based Architecture

Components communicate by sharing events. Publish/subscribe systems can be used for sharing news, balancing workloads, asynchronous workflows, etc.

In practice, many systems are hybrids of the architectures listed above.

3.3 Self-Management

Self-managing systems can be constructed using a feedback loop that monitors system behaviors and adjusts the system's internal operation.

4 Processes

4.1 Controlling Processes in Linux

Below are some useful process-related commands:

- ps - lists running processes
- top - lists processes with top resource usage
- kill / pkill / killall - used to terminate processes
- jobs - lists currently running jobs
- bg - backgrounds a job
- fg - foregrounds a job
- nice / renice - sets priority of processes
- CTRL-C - stops job in terminal
- CTRL-Z - suspends job in terminal (use fg or bg to resume)

4.2 Context Switching During IPC

Inter-process communication can be used to help coordinate processes. However, IPS can be costly, since it requires a context switch from user to kernel space and back.

4.3 Threads

Threads running in the same process can communicate through shared memory. Threads are sometimes referred to as lightweight processes (LWPs). Multithreading applications often follow a dispatcher/worker design, where a dispatcher thread receives requests and feeds them to a pool of worker threads.

4.4 Hardware and Software Interfaces

Processes and threads may interact with underlying hardware either directly through processor instructions or indirectly through library functions or operating system calls. The general layers of interaction (from more abstract to less) are applications, libraries, operating system, and hardware. These layers may interact with any layer below. Distributed systems often run in virtual environments that manage interactions with lower layers. A major benefit of such virtualization is improved portability. Virtual machine monitors can offer additional benefits, including server consolidation, live migration of VMs to support load balancing and proactive maintenance, and VM replication.

4.5 Interfaces in Network Systems

Networked applications communicate by exchanging messages. The format of these messages is determined by a message-passing protocol.

4.6 Server Clusters

Servers in a cluster are often organized in three physical tiers. The first is a load balancer, followed by application servers followed by a database or file system.

5 Communication

5.1 Access Transparency

Middleware can be used to provide access transparency for a distributed system, meaning that differences in data representation and how data is accessed can be hidden. This is typically done by middleware isolating the application layer from the transport layer.

5.2 Remote Procedure Calls (RPCs)

RPCs serve as the equivalent of conventional procedure calls, but for distributed systems. RPCs are implemented using a client-server protocol, where a client interacts with a network using a piece of software known as a client stub, and a server interacts with a network using a server stub. The steps of a typically RPC are detailed below:

1. Client process invokes client stub using ordinary procedure call.
2. Client stub builds message, passes it to client OS.
3. Client OS sends message to server OS.
4. Server OS delivers message to server stub.
5. Server stub unpacks parameters, invokes appropriate handler in server process.
6. Handler processes message, returns result to server stub.
7. Server stub packs result into message, passes it to server OS.
8. Server OS sends message to client OS.
9. Client OS delivers message to client stub.
10. Client stub unpacks result, delivers it to client process.

5.2.1 Representation of Values

Packing parameters into a message is known as parameters marshalling (with unpacking known as demarshalling). Although data representations can be different across multiple machines (e.g. big-endian vs little-endian representation), the purpose of marshalling is to represent data in a machine-independent and network-independent format that all communicating parties expect to receive. The signatures of RPC calls can be defined using what is known as an interface definition language (IDL).

5.2.2 Synchronous vs Asynchronous

In a synchronous RPC, the client waits for a return value from the server before resuming execution. In an asynchronous RPC, the client may resume execution as soon as the server acknowledges receipt of the request (client does not need to wait for a return value). A variation of an asynchronous request where a client does not wait to receive any acknowledgment from the server is known as a one-way RPC.

5.3 Message Queuing Model

As an alternative to RPCs, components in a distributed system may also communicate using a message queue that persists messages until they are consumed by a receiver. This technique allows for persistent communication that does not need to be tightly coupled in time. The primitive actions of a simple message queue are:

- **Put:** Append message to queue.
- **Get:** Block until queue not empty, then remove first message.
- **Poll:** Check specified queue for messages, remove the first message. Never block.
- **Notify:** Install handler to be called when message put into queue.

A key disadvantage of message queuing is that the delivery of the message ultimately rests with the receiver, and often cannot be guaranteed. Message queuing follows a design similar to publish-subscribe architectures, and is an example of message-oriented middleware (MOM).

5.4 Coupling Between Communicating Processes

5.4.1 Referential Coupling

Referential coupling means that one process has to explicitly reference another one for them to communicate (e.g. connecting to web server using IP address and port number).

5.4.2 Temporal Coupling

Temporal coupling means that processes must both be running for them to communicate (e.g. client cannot execute RPC if server is down).

6 Apache Thrift

6.1 Overview

Apache Thrift is an IDL first developed by Facebook, and now managed as an open-source project in the Apache Software Foundation. Thrift provides a software stack and code generation engine to support RPCs between applications written in a wide variety of common languages, including C++, Java, Python, and PHP.

6.2 Thrift Software Stack

The elements in the Thrift software stack, from top to bottom, are as follows:

- **Server:** Sets up the lower layers of the stack, and then awaits incoming connections, which it hands off to the processor. Servers can be single or multi-threaded.
- **Processor:** Handles reading and writing IO streams, and is generated by the Thrift compiler.
- **Protocol:** Defines mechanism to map in-memory data structures to wire format (e.g. JSON, XML, compact binary)
- **Transport:** Handles reading to and writing from network (e.g. using HTTP, raw TCP, etc)

6.3 Distribution Transparency

If Thrift clients must know the hostname and port for a given service, location transparency is violated. Also, although IDLs seek to provide access transparency, that too may be violated in certain conditions (e.g. when certain protocol or transport exceptions are thrown).

6.4 Application Protocol Versioning

Thrift fields can be modified and remain compatible with old versions, provided that the rules below are followed:

- Fields are associated with tag numbers, which should never the changed.
- New fields must be optional and provide default values.
- Fields no longer needed can be removed, but their tag numbers cannot be reused.

6.5 Some Programming Tips

- To separate policy from mechanism, it is generally a bad idea to hardcode hostnames and port numbers; it is usually preferable to accept command line arguments or use a property file.
- Objects built on top of TCP/IP connections (e.g. TSocket, TServerSocket) should be reused if possible to avoid to overhead of establishing and tearing down connections.
- By default Thrift transports, protocols and client stubs are not thread safe; different threads should share these items carefully, or not share them at all.

7 Distributed File Systems

7.1 Modes of Access

- **Remote access model:** Client sends requests to access file stored on remote server.
- **Upload/download model:** File is downloaded from remote server, processed on client, and uploaded back to the server.

7.2 Network File System (NFS)

NFS is a DFS first developed by Sun Microsystems in 1984, and remains heavily used in Unix-like systems. Features include:

- Client-side caching to reduce client-server communication
- Delegation of files from a server to a client, where a client can temporarily store and access a file, after which the delegation is recalled and the file returned.
- Compound procedures (e.g. lookup file, open file, and read data all in one call instead of 3). Note that NFS uses RPCs internally.
- Exporting different parts of a file system to different remote servers.

7.3 Google File System (GFS)

GFS is a DFS that stripes files across multiple commodity servers, and is layered on top of ordinary Linux file systems. Although GFS is proprietary, the Hadoop Distributed File System (HDFS) is a simplified open-source implementation of GFS. Below are a few key properties of GFS:

- **Synchronization:** A GFS master stores data about files and chunks. This metadata is cached in the main memory of chunk servers, and updates to these chunks servers are written to their own main memory. The master periodically polls the chunk servers to keep this metadata consistent.
- **Reads:** For reads, a client sends a file name and chunk index to the master, which responds with an address for the server storing that chunk.
- **Writes:** Updates are written directly to chunk servers, after which the changes are propagated through all primary and then secondary replicas.

7.4 File Sharing Semantics

In centralized file systems, reads and writes are strictly ordered in time. This ensures that an application can always read its own writes. This behavior is often described as UNIX semantics, and can be attained in a DFS so long as there is a single file server and files are not cached.

When a cached file in a DFS is modified, it is impractical to immediately propagate the changes back to the remote server (this would largely defeat the purpose or caching in the first place). Instead, a widely implemented rule is that changes are only visible to the process or machine that modified a file, and only made visible to other processes or machines when the file is closed. This rule is known as session semantics.

Semantics of DFS can be defined using combinations of various techniques. For example, NFSv4 supports session semantics and byte range file locking, and HDFS provides immutable files along with an append function (e.g. for storing log data).

8 Hadoop MapReduce

8.1 Background

MapReduce frameworks address the problem of parallelizing computations on big data sets across many machines. Google did much of the initial work on a generic MapReduce framework intended to phase out the need for special-purpose frameworks for different kinds of computations. Hadoop MapReduce is the most prominent open-source implementation of Google's MapReduce framework, and was created by Yahoo engineers Doug Cutting and Mike Cafarella.

8.2 High-level Architecture

Hadoop consists of a MapReduce engine and an HDFS system. The MapReduce layer contains a JobTracker, to which clients can submit MapReduce jobs. This JobTracker pushes work to available TaskTracker nodes, with the goal of picking a node close to the data that job requires.

8.3 MapReduce Basics

A few key guiding principles of MapReduce are:

- Components do not share data arbitrarily, as the communication overhead for keeping data synchronized across nodes can be very high.
- Data elements are immutable. Computation occurs by generating new outputs, which can then be forwarded into the next computation phase.
- MapReduce transforms lists of input data elements into lists of output data elements. A single MapReduce program typically does so twice, once for a Map phase and once for a Reduce phase.

8.3.1 Mappers

A list of data elements is provided to a mapper, which transforms each element to some output element.

8.3.2 Reducers

A reducer receives an iterator of input values, and combines these values together to produce a single value.

8.4 Technical Definitions

- **InputSplit:** A unit of work assigned to a single map task.
- **InputFormat:** Determines how input files are parsed, which also determines the InputSplit.
- **RecordReader:** Loads data from input split, creating key-value pairs used by the mapper.
- **Partitioner:** Determines which partition key-values pairs should go to.
- **OutputFormatter:** Determines how output files are formatted.
- **RecordWriter:** Writes records to output files.

8.5 Fault Tolerance

The main way Hadoop achieves fault tolerance is by restarting failed tasks. TaskTracker nodes are in constant communication with the JobTracker node, so should a TaskTracker fail to respond in a given period of time, the JobTracker will assume it has crashed. If the job that crashed was in a mapping phase, then other TaskTracker nodes will receive requests to re-run all map tasks previously run by the failed node. If the job that crashed was in a reduce phase, then other TaskTracker nodes will receive requests to re-run all reduce tasks that were in progress on the failed node.

Such a simple fault tolerance mechanism is possible because mappers and reducers limit their communication with one another and the outside world. A potential drawback with such an approach is that a few slow nodes (called stragglers) can create bottlenecks that hold up the rest of the program. One strategy to remedy this problem is known as speculative execution, where the same task is assigned to multiple nodes to decrease the expected time in which it will be finished.

8.6 Common Programming Patterns

8.6.1 Counting and Summing

The simplest mapper for a counting problem would output a (key, 1) tuple for an instance of a given key. Alternatively, the mapper can aggregate data for an entire document, or a combiner can be run just after the mapper to aggregate data across documents processes by a map task in a similar way to how a reducer would.

8.6.2 Selection

Selection returns a subset of input elements that meet a certain requirement. Selection can be handled entirely in the map task; a reducer need not be specified, in which case one output file will be generated per map task.

8.6.3 Projection

Projection returns a subset of fields of each input element (e.g. a, b, c becomes a, b). A mapper can handle projection on individual elements, while a reducer is only needed to eliminate duplicates.

8.6.4 Index Inversion

Index inversion produces a mapping of terms to document ids. Mappers emit (term, id) tuples, while reducers combine these tuples to generate lists of ids for each term.

8.6.5 Cross-Correlation

Cross-correlation problems provide as input a set of tuples of items, and for each possible pair of items, the number of items where the tuples co-occur is measured. If N items are provided, then N^2 values should be reported. This problem can be represented through a matrix if N^2 is small enough. For larger values of N^2, MapReduce can be effective.

A pairing implementation where mappers return tuples with a pair and a 1 count is simple but often not performant. A "stripes" approach where mappers return tuples with an item and a list of items it appears with tends to perform better despite requiring more memory for map-side aggregation, since there end up being roughly N keys (one for each element) instead of N^2 (one for each pair).

9 Apache Spark

9.1 Background

Cluster computing frameworks (e.g. Hadoop) allow for large-scale data computations that automatically handle work distribution and fault tolerance. However, many of these frameworks do not leverage distributed memory efficiently, making them ineffective for computations with intermediate results. Apache Spark introduces the concept of resilient distributed datasets (RDDs), which are parallel, fault-tolerant data structures that persist intermediary results in memory.

9.2 Lineage

A lineage is a model of the flow of data between RDDs. Spark designs a lineage to perform efficient computations, and also uses the lineage to determine which RDDs to rebuild to recover from failures.

9.3 Transformations and Actions

Transformations are data operations that convert an RDD or a pair of RDDs into another RDD. Actions are data operations that convert an RDD into an output. When an action is invoked on an RDD, the Spark scheduler puts together a DAG of transformations.

9.3.1 Narrow vs Wide Dependencies

The transformations in the DAG are grouped into stages. A single stage is a collection of transformations with narrow dependencies, meaning that one partition of the output depends only on a single partition of the input (e.g. map, filter). The boundaries between stages feature wide dependencies, meaning that a single partition of output may correspond to multiple partitions of input (e.g. group by key), so the transformations may require a shuffle.

9.4 Spark vs Hadoop MapReduce

Two common differences between Spark and MapReduce are:

- Spark stores intermediary results in memory, while MapReduce dumps results to HDFS between each job. The MapReduce approach can lead to unnecessary I/O when running multiple jobs in sequence.
- Spark can support some more complicated workflows within a single job rather than running several new ones.

10 Distributed Graph Processing

Many data sets (e.g. web hyperlinks, social networks, transportation routes) can be modeled using graphs. Computations concerning very large graphs can benefit from distributed architectures. However, previously examined cluster computing frameworks (e.g. Hadoop MapReduce, Apache Spark) are typically not good fits for graph processing, which motivates the creation of separate distributed graph processing frameworks.

10.1 Google's Solution: Pregel

Pregel is a proprietary framework developed by Google to perform computations on large distributed graphs with performance and ease of programming in mind. A master/worker model similar to those of MapReduce and Spark is used, where each worker is responsible for a particular vertex partition. The framework maintains some state information for each vertex, including:

- Problem-specific values
- List of messages sent to vertex
- List of outgoing edges from a vertex
- A binary active/inactive state

10.1.1 Supersteps

Pregel applies a bulk synchronous parallel (BSP) model of computation that where a computation is divided into iterative rounds called supersteps. Workers perform their computations asynchronously within each superstep, and only exchange data between supersteps. Specific actions that can be taken by a worker within a superstep include:

- Receiving messages sent in the previous superstep.
- Sending messages to be received in the next superstep.
- Modifying vertex values or edges.
- Deactivating a vertex, which is reactivated when it receives a message.

This distributed execution stops when all vertices are inactive and with no more messages to process.

10.1.2 Initialization

The master is responsible for assigning a section of the vertices to each worker. The default partitioner uses a simple hash function over vertices, which generates a fairly even distribu-

tion of vertices. To take advantage of other properties of a graph (e.g. exploiting locality), the default partitioning scheme can be overridden.

10.1.3 Combiners and Aggregators

Pregel supports combiners, which serve to reduce the amount of data exchanges over a network and the number of messages. Combiners are often applicable when the function applied at each vertex is commutative and associative (e.g. sum, max).

Pregel also supports aggregators, which are used to compute aggregate statistics from vertex-reported values.

10.1.4 Fault Tolerance

At the end of each superstep, the master instructs workers to save their state (i.e. vertex values, edge values, incoming messages, aggregator values, etc.) to persistent storage. When the master detects the failure of a worker node, it rolls back all workers to the last successful superstep, and the computations resume. If deterministic replay of sent messages is possible, then there are some more efficient recovery mechanisms that only require that the failed worker is rolled back to the last successful superstep.

10.2 Open-source Implementations

Pregel-like APIs are supported by various open-source frameworks such as Apache Giraph. Other APIs support centralized graph processing (e.g. GraphChi), which tends to perform better for fairly small datasets that can fit in the main memory of a single machine.

11 Consistency and Replication

11.1 Replication Overview

11.1.1 Purposes of Replication

The main purposes behind data replication include:

- **Increasing reliability**, since data is more likely to be accessible if it exists on more replicas.
- **Increasing throughput**, since replicas can serve read operations in parallel.
- **Lowering latency**, since replicas physically close to clients can avoid costly round trips.

11.1.2 Replicated Data Stores

In replicated data stores, each data object is replicated across multiple hosts. Replicas may reside on the same host as processes that interact with them, or may be remote.

11.2 Consistency Models

Consistency models describe the extent to which replicas can disagree on the current state of data. Managing consistency tends to be straightforward for read-only data, but the need for sophisticated models arises once replicas hold mutable states, especially ones shared among multiple processes. Selecting a good consistency model is often challenging, as application requirements rarely map neatly to a specific model.

11.2.1 Sequential Consistency

Sequential consistency requires that:

1. The result of any execution is the same as if the operations by all processes on the data store were performed in some sequential order.
2. The operations of each individual process appear in the order specified in its program.

The notion of this consistency model is borrowed from shared memory multiprocessing.

11.2.2 Causal Consistency

The concept of causal consistency is also borrowed from shared memory multiprocessing, and is based on a notion of similarity similar to Lamport's "happens before" model. A data

store is casually consistent if writes related by some "causally precedes" operation are seen by all processes in the same order. The "causally precedes" operator is defined as follows:

1. Op1 causally precedes Op2 if Op1 occurs before Op2 in the same process.
2. Op1 causally precedes Op2 if Op2 reads a value written by Op1.

11.2.3 Linearizability

A data store is linearizable when the result of any execution is the same as if all operations on the data store were executed in some sequential order that extends Lamport's "happens before" model. This means that if Op1 finishes before Op2 beings, then Op1 must precede Op2 in sequential order.

11.2.4 Eventual Consistency

The idea behind eventual consistency is that if no updates take place for some time, all replicas will gradually become consistent. Eventual consistency allows for different processes to observe write operations taking effect in different orders, even when the operations are related by "causally precedes" or "happens before".

Eventual consistency is a very weak property on its own, and appears to guarantee very little if updates are applied continuously. To strengthen these guarantees, session guarantees can be applied, which restrict the behavior of a single process in a single session. Examples of session guarantees include:

- **Monotonic reads**, which means that if a process reads a particular value of a data item, then successive reads in that process will return that value or a newer one.
- **Read your own writes**, which means the effect of a write operation will always be seen in successive read operations by the same process.

11.3 Replication Protocols

11.3.1 Primary-Based Replication Protocols

Primary-based replication protocols involve updates being performed on a single primary replica, with updates then pushed to any backup replicas. If a primary replica is stationary, where it may be often updated remotely by other servers, this protocol can be classified as remote-write. Alternatively, local updates may be enabled by allowing the primary replica to migrate from server to server, in which case the protocol can be classified as local-write.

Primary-based replication protocols allow for some strong consistency models. However, a single primary replica can lead to performance bottlenecks and availability problems.

11.3.2 Quorum-Based Replication Protocols

Quorum-based replication protocols allow all replicas to perform updates and to service reads. However, each update and read must be performed on a sufficiently large subset of replicas, known as the write and read quorum respectively.

Let N be the total number of replicas, N_R be the read quorum, and N_W be the write quorum. In a distributed database, the following two rules must be satisfied:

1. $N_R + N_W > N$, which enables the detection of read-write conflicts.
2. $N_W + N_W > N$, which enables the detection of write-write conflicts.

Subsets of replicas that do not satisfy the rules above are referred to as partial quorums. Alternate means must be found for resolving conflicts, such as tagging updates with timestamps.

11.3.3 Eventually-Consistent Replication Protocols

In simple cases of eventual consistency, reads and updates are serviced by the closest replica. This replica lazily (i.e. async to original update) propagates updates to other replicas. Replicas can go out of sync for various reasons, such as one being unreachable due to network failure. This is handled by what is known as an anti-entropy mechanism. One sample anti-entropy mechanism is different replicas exchanging hash trees (also known as Merkle trees), which contain hashes of blocks of data that can be used to find where two data sets differ.

12 Fault Tolerance

12.1 Dependability

Fault tolerance is closely related to the concept of dependability, which implies the following requirements:

- **Availability:** The system should operate correctly at a given instant in time (e.g. 99% availability means a system operates correctly 99% of the time).
- **Reliability:** The system should run continuously without interruption (e.g. mean time between failures (MTBF) of one month).
- **Safety:** System failures should not have catastrophic consequences.
- **Maintainability:** A failed system should be easy to repair.

12.2 Failures

A failure is occurring when a system cannot fulfill its promises. Five major types of failures of servers in distributed systems are:

- **Crash failure:** A server is working correctly and then halts.
- **Omission failure:** A server fails to respond to requests or fails to receive or send messages.
- **Timing failure:** A server's response falls outside of some specified time window.
- **Response failure:** A server's response is incorrect.
- **Arbitrary failure:** A server produces arbitrary failures at arbitrary times.

12.3 Errors and Faults

An error is a part of a system's state that may lead to failure (e.g. corrupt data read from hard disk). The fault is the underlying cause of the error (e.g. hard disk head crashes). The main types of faults are:

- **Transient faults,** which emerge once and then disappear (e.g. birds flying in front of a microwave receiver)
- **Intermittent faults,** which tend to reappear (e.g. loose contact on electrical connector)
- **Permanent faults,** which continue to exist until some faulty component is replaced (e.g. burned out power supply)

12.4 Masking Failure by Redundancy

Triple-modular redundancy (or more generally, N-modular redundancy) is an approach to fault tolerance where multiple (3 or N) subsystems perform the same process, after which the result is determined by a majority voting system. Thus, the system can continue to function if some of its subsystems fail so long as the remaining systems can win a majority vote. To support voting, processes must be able to communicate with one another. This communication can be setup as a flat group (i.e. complete graph between all processes) or in some sort of hierarchical structure with a coordinator process and multiple worker processes.

12.5 Consensus Problem

The consensus problem in distributed computing can be defined as follows:

- Each process has procedures propose(val) and decide()
- Each process first proposes a value by calling propose(val) once
- Each process learns the value agreed upon by calling decide()

The following properties must hold:

- **Agreement:** Two calls to decide() cannot return different values.
- **Validity:** If decide() returns a value val, then some process called propose(val).
- **Liveness:** Calls to decide() and propose(val), if a process does not fail, must eventually terminate.

12.5.1 Variations

The solvability of the consensus problem depends on several factors of a distributed environment, including:

- Whether processes are synchronous or asynchronous to one another
- Communication delays between processes
- Message delivery order
- Whether messages are unicast (one-to-one) or multicast (one-to-many)

12.6 RPC Semantics Under Failures

RPC systems may exhibit the following failure scenarios, which depend on when a failure occurs in an RPC call:

- Client unable to locate server.

- Request message from client to server is lost.
- Server crashes after receiving request.
- Reply message from server to client is lost.
- Client crashes after sending request.

12.6.1 Dealing with RPC Server Crashes

When the server crashes after receiving a request, the client may not know whether the server crashed before or after executing the request. Techniques for dealing with such RPC server crashes include:

- Reissuing the request, leading to at-least-once semantics, which are safe as long as the request is idempotent (i.e. repeated executions have the same effect as one execution).
- Not reissuing the request and reporting a failure, leading to at-most-once semantics, with no guarantee that the request was processed.
- Determining whether the request was processed, and if it was not, reissuing the request. This leads to exactly-once semantics, which may be difficult to implement as there may be no clear way of knowing what action was performed.

13 Apache ZooKeeper

ZooKeeper is a centralized system that manages distributed systems as a hierarchical key-value store. Common uses of ZooKeeper include maintaining configuration information, providing distributed synchronization, and managing naming registries for distributed systems. ZooKeeper emphasizes good performance (particularly for read-dominant workloads), being general enough to be used for many different purposes, reliability, and ease of use.

13.1 Data Model

ZooKeeper's data model is a hierarchical key-value store similar to a file system. Nodes in this store are called znodes, and may contain data and children. Reads and writes to a single node are considered to be atomic, with values read or written fully or not at all.

13.2 Node Flags

Two important flags nodes can carry are:

- **Ephemeral flags**, which make nodes exist as long as the session that created them is active, unless they were explicitly deleted.
- **Sequence flags**, which make nodes append a monotonically increasing counter to the end of their path.

13.3 Consistency Model

ZooKeeper ensures that writes are linearizable and that reads are serializable (a similar property to sequential consistency). ZooKeeper also guarantees per-client FIFO servicing of requests.

13.4 Servers

When running in replicated mode, all servers have a copy of the state in memory. A leader is elected at startup, and all updates go through this leader. Update responses are sent once a majority of servers have persisted the change. Thus, in order to tolerate n failures, $2n + 1$ replicated servers are required.

14 Distributed Commits and Checkpoints

14.1 Transactions

Transactions are indivisible operations with the following properties:

- **Atomic:** An operation occurs fully or not at all. This can be hard to achieve in distributed environments.
- **Consistent:** A transaction is a valid transformation of the state.
- **Isolated:** A transaction is not aware of other concurrent transactions.
- **Durable:** Once a transaction completes, its updates persist, even in the event of failure.

14.2 Two-Phase Commits

The two-phase commit (2PC) is a distributed transaction commitment protocol. Systems running this protocol consist of a coordinator and participants.

In the first phase, the coordinator asks participants whether they are ready to commit. These participants respond by voting. In the second phase, the coordinator analyzes the participant votes. If all participants voted to commit, the commit proceeds. Otherwise, the transaction is aborted.

The 2PC procedure described above makes a few key assumptions, including that processes are synchronous and that communication delays are bounded.

14.2.1 Recovery from Failures

Participants are able to make progress so long as they have received a decision. This decision is typically received from the coordinator, though in the case of a coordinator crash, may be received from another participant.

The protocol described above blocks when all participants are waiting for an answer and the coordinator crashes, though some more complicated implementations can overcome this. However, even in these more complicated implementations, the simultaneous failure of the coordinator and some participants can make it difficult to detect whether all participants are ready for an answer.

14.2.2 Distributed Checkpoints

Recovery after failure is only possible if the collection of checkpoints by individual processes forms what is called a distributed snapshot. A distributed snapshot requires that process

checkpoints (i.e. representations of state at a certain point) contain a corresponding send event for each message received. The most recent distributed snapshot is called the recovery line.

Coordinated checkpointing algorithms can be applied to create recovery lines. A sample two phase checkpointing algorithm works as follows:

1. The coordinator sends a checkpoint request message to all participant processes. When a participant receives the message, it pauses processing of incoming messages, forms a checkpoint, and returns an acknowledgment (ACK) to the coordinator.
2. Once all ACKs are received by the coordinator, it sends a message to the participants that they can resume processing incoming messages.