Distributed Systems

There are 2 hard problems:

2) Exactly-once delivery

1) Guaranteed order of messages

2) Exactly-once delivery

Leon Muscat

Contents

0 Module Description

0.1 Learning Objectives

- Explain, compare, and apply architectural styles for distributed systems: client/server, implicit invocation, peer-to-peer, mobile code, etc.
- Understand and discuss fundamental topics: fault-tolerance, replication, consensus, consistency, and logical time.
- Have an in-depth understanding of the Web Architecture and engineer complex Web-based systems.
- Understand industry-relevant breakthroughs in distributed systems: Google File System, MapReduce, Spark.
- Design and program scalable, fault-tolerant, and adaptable distributed systems.

0.2 Event Structure and Teaching/Learning Design

- Interactive lectures with in-class exercises and graded practical exercises.
- Topics covered: network-based architectures, consistency and logical time, fault-tolerance and consensus, distributed storage, computing with distributed datasets, decentralization and blockchain, distributed intelligence.

0.3 Literature

- M. van Steen and A.S. Tanenbaum: Distributed Systems, 4th ed., 2023.
- R.N. Taylor, N. Medvidovic, E.M. Dashofy: Software Architecture: Foundations, Theory, and Practice, 2010.
- Additional literature from lectures and exercises.

0.4 Examination

0.4.1 1st Examination Component (1/2)

- **Type**: Analog written examination
- **Form**: Written exam
- **Mode**: Analog
- **Time**: Lecture-free period
- **Location**: On Campus

- **Grading**: Individual work, individual grade
- **Weight**: 60%
- **Duration**: 90 Min.
- **Languages**: Questions in English, Answers in English.
- **Aids**: Closed Book, specific models of the Texas Instruments TI-30 series allowed, bilingual dictionaries (without notes) allowed for non-language exams.

0.4.2 2nd Examination Component (2/2)

0.4.2.1 Remarks

- **Type**: Programming
- **Form**: Practical test
- **Mode**: Digital
- **Time**: Lecture time
- **Location**: Off Campus
- **Grading**: Group work, group grade
- **Weight**: 40%
- **Languages**: Questions in English, Answers in English.
- **Aids**: Free aids provision with any restrictions defined by faculty.

0.4.2.2 Tools and Materials

- Central Exam (60%): All topics from lectures, exercises, discussions, and referenced literature.
- Exercises (40%): Graded exercises based on specific lecture topics.

0.4.2.3 Examination Content

- Central Exam: All topics from lectures and exercises.
- Exercises: Graded exercises from specific lecture topics.

0.4.2.4 Examination Literature

- Lecture and exercise materials, referenced literature, and discussions from lecture and exercise sessions.

1 Introduction to Distributed Systems

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

- Leslie Lamport, 1987

More precisely, a distributed system is a collection of networked computers systems, which work together to achieve a coherent task, while processes and resources are spread across different computers.

Every large scale web infrastructure we use is dependent on the technology of distributed systems.

1.1 Design goals

The objective of a distributed system is to find abstractions and techniques that mitigate the complexity of the distribution. There are three main aspects to consider for this objective: Storage, communication, and computation.

1.1.1 Scalability

Scalability is the ability of a system to be adapted to meet new requirements of size and scope.

Scaling can be achieved by hiding communication latencies (via asynchronous communication), partition data and compution across multiple machines, and replication and caching.

1.1.1.1 Size Scaling We differentiate between vertical scaling, which means adding more resources to a computer, and horizontal scaling, which means adding more computers.

Ideally, $n \times$ resources $\rightarrow n \times$ throughput, but this is rarely the case due to the following reasons:

- The computation capacity, limited by the CPUs
- The storage capacity, including the transfer rate between CPUs and disks
- The network bandwith, e.g. between the user and the service

1.1.1.2 Geographical Scaling Geographical scaling refers to increasing the maximum distance between nodes.

This introduces issues, such as geographical boundaries, high latency, and unreliable connections.

1.1.1.3 Administrative Scaling This refers to an increase in the number of administrative domains, and introduces issues, such as trust, security, fragmented change, and policy.

1.1.2 Fault-tolerance

Fault-tolerance is the ability of a system to continue operating in the event of a failure within some of its componenets.

Facets of fault-tolerance:

- **Availability**: The property that a system is ready to be used immediately
- **Reliability**: The property that a system can run continuously without failure
- **Safety**: The property of a system that when it temporarily fails to operate correctly, no catastrophic event happens
- **Mainability/Recoverability**: The property of a system to be easily repaired once failed

Fault-tolerance can be achieved through use of non-volatile storage for recovery checkpoints, and replication.

1.1.3 Data Consistency

Data consistency refers to whether the same data kept at different places is kept in sync.

There are models of consistency:

- **Strong consistency**: A read always retrieves the value of the most recent write. This is used when immediate consistency is necessary, but is more expensive (e.g. flight booking).
- **Eventual consistency**: A read retrieves a value that was previously written, and will eventually retrieve the most recent one. This is used when the exact number is not important, but performance is (e.g. social media).

1.1.4 Distribution Transparency

Distribution transparency refers to whether the distribution of processes and resources is visible to end users and application developers. Full transparency means the user does not know he is communicating with a distributed system.

There are different types of transparency:

- **Access**: Hide differences in data representation and how an object is accessed
- **Location**: Hide where an object is located
- **Relocation**: Hide that an object may be moved to another location while in use
- **Migration**: Hide that an object may move to another location
- **Replication**: Hide that an object is replicated

- **Concurrency**: Hide that an object may be shared by several independent users
- **Failure**: Hide the failure and recovery of an object

Full transparency is not possible:

- There are always communication latencies
- Completely hiding failures of networks and nodes is impossible
- Full transparency will cost performance, thus exposing the distribution of the system
- Exposing distribution may be a good thing

1.2 Challenges

There are 2 fundamental Challenges for designing distributed systems:

- **Concurrency**: There are usually many components with complex interactions among them
- **Partial failures**: Network communication may fail, processes may crash, all of which may happen nondeterministically

This introduces the CAP Theorem, which states that any networked computer system providing shared data can have at most 2/3 desirable properties:

- **Consistency**, by which a shared and replicated data item appears as a single, up-to-date entry
- **Availability**, by which updates will always be eventually executed
- Tolerant to network **partitions**. A partition is a division of a distributed network into relatively independent subnets.

These challenges are due to the following false assumptions:

- The network is reliable (account for network failures)
- The network is secure (account for security threats)
- The network is homogeneous (account for heterogeneity)
- The topology does not change (design for evolution)
- Latency is zero (be mindful of network latency)
- Bandwith is infinite (be mindful of bandwith usage)
- Transport cost is zero (network communication is costly)
- There is one administrator (be mindful of devops)

In practice, it is all about **design trade-offs**: distributed database management tends to prioritize either consistency or availability, but modern systems usually achieve flexible trade-offs between the two

The challenge is to **mitigate partitions effectively**:

- Detect the start of a partition
- Enter an explicit partition mode that may limit some operations
- Initiate partition recovery when communication is restores (restore consistency)

2 MapReduce

MapReduce is a programming model and associated implementation introduced by Google for processing and generating large datasets.

2.1 Overview

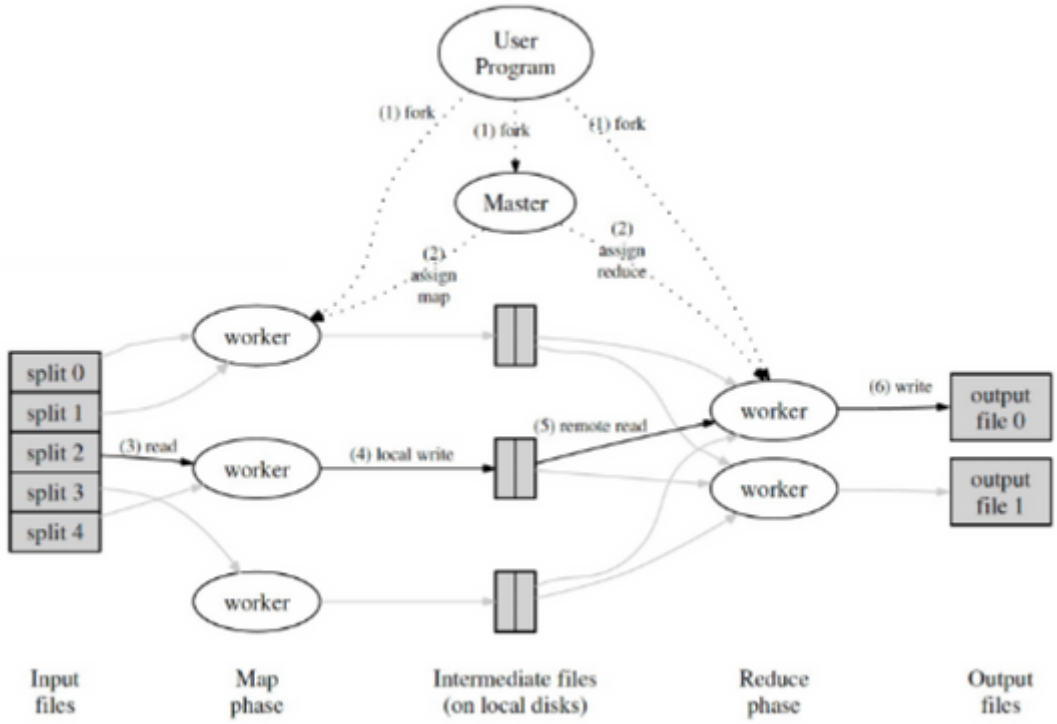


Figure 1: MapReduce

- Developed to handle massive amounts of data.
- A simple and flexible interface for large scale computation across many machines.
- Especially useful for large scale data processing tasks, like search indexing and data mining.

2.2 Key Concepts

2.2.1 1. Map

- The `Map` task takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).

2.2.2 2. Reduce

- The `Reduce` task takes the output from the `Map` as input and combines those data tuples (based on the key) into an aggregated set of tuples.

2.3 Workflow

1. **Input Splitting**: Large data sets are split into smaller chunks. Each chunk is processed by a separate Map task.
2. **Map Step**: Each Map task processes a chunk and produces a set of key-value pairs.
3. **Reduce Step**: Each Reduce task processes a group of key-value pairs with the same key.

Both the map and reduce tasks can be distributed in a distributed system.

2.4 Advantages

- **Scalability**: Can process petabytes of data.
- **Fault-tolerance**: Tasks are automatically retried upon failure, and data is replicated.
- **Flexibility**: Can process structured and unstructured data.

2.5 Limitations

- Not suitable for all problems, especially those that require iterative computations (e.g., many machine learning algorithms).
- Overhead from splitting and organizing data can be high for certain tasks.

2.6 Fault-tolerance

Handling worker failures:

- The master pings every worker periodically to detect failures

- If a map worker fails, completed and in-progress map tasks of that worker are re-executed
- If a reduce worker fails, in-progress reduce tasks are re-executed

Re-execution is necessary because the results of the failed map worker is missing.

Handling master failure (master failures are less likely):

- Can be handled through recovery checkpoints written to non-volatile memory

2.7 Real-world Applications

- Search engine indexing.
- Data warehousing.
- Log file analysis.

3 The Representational State Transfer (REST)

The Representational State Transfer (REST) architecture, introduced by Roy Fielding in his 2000 doctoral dissertation, has become the de facto standard for building web services.

3.1 REST Architectural Constraints

REST is defined by a set of constraints that, when applied as a whole, produce a system that is scalable, performant, and maintainable. These constraints create a unique architectural style that is different from other approaches like Remote Procedure Call (RPC) or Simple Object Access Protocol (SOAP).

What are architectural constraints?

> An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.

Architectural styles limit designers by imposing constraints on architectural elements. This allows the developer to focus on the essentials, rely on invariants, and boost productivity.

Now, lets look at the constraints of REST.

3.1.1 1. Client-Server

The client-server constraint separates concerns between the client and the server, thus promoting system scalability and the independent evolution of components.

- **Client**: Responsible for the user interface and user experience. The client remains lightweight since it does not need to store any data necessary to handle requests.
- **Server**: Stores data and implements functionality. This clear separation allows for server-side scalability and independent evolution of server-side logic.

3.1.2 2. Stateless

Each request from a client contains all the information needed for the server to understand and process the request. This means that no session state is stored on the server between requests.

- **Benefits**:
  - Scalability: Without the need for storing session state, it's easier to scale out applications.
  - Reliability: If a request fails, the client can simply retry, as there's no session state that might be left in a problematic state.
- **Drawbacks**:
  - Overhead: Sending all the necessary information with every request can lead to larger payloads.

3.1.3 3. Cache

To improve performance, responses from the server can be explicitly marked as cacheable or non-cacheable. If a response is cacheable, the client cache can reuse it for equivalent requests in the future.

- **Benefits**:
  - Efficiency: Reduces the need for repeated requests to the server.
  - Scalability: Reduces the load on the server.

3.1.4 4. Uniform Interface

A key feature of REST is the uniformity of its interface, which simplifies the architecture and helps decouple the client and server. This constraint is defined by four sub-constraints:

- **Resource-Based**: Individual resources are identified in requests using URIs, and resources themselves are conceptually separate from the representations sent to the client.
- **Manipulation of Resources Through Representations**: Clients interact with resources by exchanging representations (like JSON or XML).
- **Self-Descriptive Messages**: Each message contains enough information to describe how to process the message.
- **Hypermedia as the Engine of Application State (HATEOAS)**: Clients navigate the state of the app through hypermedia links provided dynamically by the server in the responses.
- **Benefits**:
  - Evolvability: Decouples the architecture and allows components to evolve independently of one another
  - Simplicity: Simplifies the overall system.
  - Visibility: Easier to monitor interactions in the system.
- **Drawbacks**:
  - Performance: A general interface is not well-optimized for the specific needs of individual applications.

3.1.5 5. Layered System

In a layered system, an architecture is organized into layers, where each layer has a specific functionality. A component in a layer interacts only with the layer immediately below it, and it's unaware of the components in other layers.

- **Benefits**:
  - Modularity: Layers can be replaced or updated independently.
  - Security: Potential security breaches in one layer do not compromise other layers.

3.1.6 6. Code on Demand (Optional)

This is the only optional constraint in the REST architecture. Servers can extend the functionality of a client by transferring executable code. For instance, a server can send client-side scripts (like JavaScript) to run within a web page.

- **Benefits**:

  - Flexibility: Allows for evolving client-side functionality without having to update the client itself.

- **Drawbacks**:
  - Security: Running code from servers can introduce security risks.

3.1.7 Conclusion

REST's architectural constraints, when applied properly, lead to a system that is scalable, performant, modular, and maintainable. The constraints work in tandem to provide a cohesive approach to building web services, making REST a popular choice for developers worldwide.

These constraints encapsulate the essence of the web and are necessary for the internet-scale systems we use.

3.2 REST APIs

REST's uniform interface allows for and requires Hypermedia as the Engine of Application State, which decouples the web architecture: URIs, possible transitions to next application states, and the means to transition to those states are advertised in the hypermedia – they are never hard-coded into clients!

The same should go for REST based APIs, however, this is often not the case. A lot of APIs have tight coupling, meaning that the client needs to know the architecture of the API, in order to use it. A good example is Twitter's V1 API.

To better judge the RESTfulness of APIs, Leonard Richardson developed a classification of web services.

3.2.1 Richardson Maturity Model

The Leonard Richardson Maturity Model is a tool that helps in understanding the progression of designing web services, particularly those adhering to the REST architectural style. It breaks down the journey of implementing RESTful services into four distinct levels, moving from the lowest level of maturity (Level 0) to the highest (Level 3).

It is important to note that a higher level does not equal a "better" API. With every level, there are certain design trade-offs, which, dependent on the application, don't make sense.

3.2.1.1 Level 0: The Swamp of POX At this level, web services use a single URI and rely on a single HTTP method, typically POST. All service functionality is driven through the body of the request, often using XML (hence "POX", which stands for "Plain Old XML").

- **Characteristics**:
  - Single endpoint.
  - Little to no use of HTTP methods; everything is typically a POST.
  - Communication is often done through RPC or SOAP.

3.2.1.2 Level 1: Resources Services at this level start to leverage multiple URIs to identify individual resources or entities in the system. However, they still tend to rely on a single HTTP method.

- **Characteristics**:
  - Multiple endpoints representing individual resources.
  - Still limited use of HTTP methods, often just POST or GET.

3.2.1.3 Level 2: HTTP Verbs At this level, services not only use multiple URIs for different resources but also utilize various HTTP methods (or "verbs") to represent operations on these resources. This is where most REST API's on the web currently are.

- **Characteristics**:
  - Multiple endpoints with individual resources.
  - Use of standard HTTP methods like GET, POST, PUT, DELETE to perform CRUD (Create, Read, Update, Delete) operations.
  - More aligned with the principles of REST, but not fully RESTful yet.

3.2.1.4 Level 3: Hypermedia Controls This is the highest level of maturity. Services at this level fully embrace the REST architecture, particularly the principle of HATEOAS (Hypermedia as the Engine of Application State). Clients interact with the service guided by hypermedia links provided in the responses.

- **Characteristics**:
  - Multiple endpoints with individual resources.
  - Full use of HTTP methods.
  - Responses contain hypermedia links that guide client interactions, making the API self-descriptive.

3.2.2 Conclusion

The Leonard Richardson Maturity Model offers a roadmap for progressing from basic web services to truly RESTful services. By understanding and implementing each level, developers can better design and evolve their web services while ensuring they adhere closely to REST principles.

4 Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a powerful tool for developers who want to create distributed applications. It allows programs to cause a procedure (subroutine) to execute in another address space (commonly on another computer on a shared network) as if it were a local procedure call, without the programmer needing to explicitly code the details for remote interaction. The objective is distribution transparency.

As with most web technologies, RPC uses the client-server model and has architectural constraints.

4.1 How RPC works

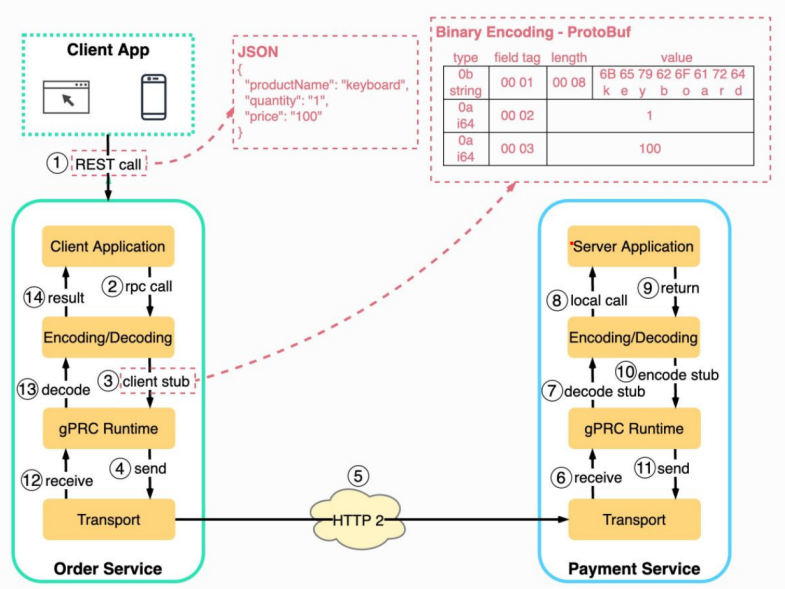


Figure 2: Illustration of RPC

1. **Stub Generation**: The client and server-side stubs are generated from the service definition (often written in an Interface Definition Language, or IDL, like ProtoBuf). These stubs act as

surrogates for the remote procedure on the client and server sides. The developers don't need to worry about the RPC implementation thanks to the generated code.
2. **Client Call**: When a client wants to make a remote procedure call, it actually calls the client-side stub.
3. **Marshalling**: The client-side stub packages (or marshals) the function parameters into a form that can be sent across the network.
4. **Communication**: The client communicates with the server using the network's communication protocol (e.g., TCP/IP).
5. **Server-Side Unmarshalling**: The server-side stub receives the request, unpacks the parameters (unmarshalling), and then calls the actual procedure on the server.
6. **Result Return**: The result is then marshalled and sent back to the client, where the client-side stub unmarshals it and passes the result back to the original caller.

4.2 Challenges

1. Complete access transparency is not possible, since processes may crash, messages may be delayed, and messages may be lost.
2. The client and server programs may be written in different programming languages using different data types and serialization formats. That means both parties need to agree on an encoding standard.

5 Interaction Styles for Web Services

Now that we understand REST and RPC, let's see how these connectors compare with RDA and EVENTful connectors.

5.1 Remote Procedure Call (RPC) Connectors

Definition: RPC connectors allow one system to cause a procedure (subroutine) to execute in another system. In the context of web services, it usually involves calling a specific method on the server.

Coffee Shop Example: A customer uses a coffee shop app to order a cappuccino. The app sends a direct call like `orderCappuccino(size, extras)` to the server, which then processes the request and returns an order confirmation.

5.2 REST Connectors (“RESTful”)

Definition: Representational State Transfer (REST) connectors operate based on standard HTTP methods (like GET, POST, PUT, DELETE) and are centered around resources rather than methods.

Coffee Shop Example: A customer chooses a cappuccino from the list of coffee resources on the app. The order is placed by sending a POST request to `/orders` with the details of the cappuccino. The server responds with a unique order ID (e.g., `/orders/1234`) which can be used later to check the status or modify the order.

5.3 Remote Data Access (RDA) Connectors

Definition: RDA connectors focus on direct access and manipulation of remote data structures, often resembling database operations.

Coffee Shop Example: A customer wants to order a cappuccino. The app accesses the coffee shop’s remote database, creates a new record in the `orders` table with the details of the cappuccino, and retrieves an order ID. This ID can be used for subsequent operations, like updating or deleting the order.

5.4 Event Connectors (“EVENTful”)

Definition: Event connectors operate on an event-driven model. Instead of directly calling methods or accessing data, operations are driven by events (HTTP Web Hooks or CoAP Obeserve), often following the publisher-subscriber pattern (W3C WebSub).

Coffee Shop Example: A customer places an order for a cappuccino through the app. This action triggers an “OrderPlaced” event. The coffee shop’s system is subscribed to this event, so when it occurs, the system starts preparing the cappuccino. When the coffee is ready, another event, “OrderReady”, is published, notifying the customer through the app that their coffee is ready for pickup.

5.5 Conclusion

The choice of interaction style largely depends on the specific requirements of the application and the context in which it operates.

- **RPC** is direct and method-focused, suitable for applications that need tight coupling between client and server.
- **RESTful** services are resource-centric and leverage the simplicity and scalability of the web’s architectural principles.

- **RDA** connectors are apt for systems that require direct manipulation of remote data structures, resembling traditional database operations.
- **EVENTful** connectors shine in asynchronous, decoupled systems where operations are driven by events.

6 Processes, Threads and Distributed Systems

An OS manages a computer's resources and provides a layer of abstraction for programs to use.

6.1 Processes

One such abstraction is a process. A process is an abstraction of a running program (a program loaded into memory along with all the resources it needs to operate).

Each process is isolated, meaning it has no contact with or effects on other processes.

That requires each process to also have its own address space, which is seperated into 4 parts:

- **Text segment**: The program code
- **Data segment**: Global variables and constants
- **Heap segment**: Memory dynamically allocated during run time
- **Stack segment**: Contains one frame for each procedure that has been entered but not yet exited

A process can create child processes to help do its tasks, but the child process has its own address space.

The OS is responsible for managing processes and scheduling them on the CPU. To do that the OS maintains a process table with an entry, called a process control block, for each process.

6.2 Threads

By default, a process only has a single thread, but in many situtations we would want to run tasks in parallel on the CPU, while using the same address space. This is where multithreaded processes come in.

Whilt the address space is shared, each thread receives its own stack within the address space.

Because threads share the same address space as the process and other threads within the process, the operational cost of communication between the threads is low, which is an advantage. The disadvantage is that a problem with one thread in a process will certainly affect other threads and the viability of the process itself.

Creating, context switching, and desctruction of threads is also much cheaper than for processes.

The thread scheduler multiplexes threads on the CPU and decides which threads can run based on the states of all threads.

6.3 Processes and threads in the context of Distributed Systems

In distributed systems, we want to make use of multiprocessing and multithreading for nomoures reasons.

6.3.1 Hide network latency

- Clients may send multiple (blocking) requests in parallel
  - If the requests are sent to different servers, we get linear speed-up
- Servers may handle multiple requests in parallel (each request may block)
  - Ex: while waiting for the disk to read data for client X, process request from client Y

6.3.2 Performance

- Execute code in parallel on multiprocessors and multicore processors

6.3.3 Programming Convenience

- For example, in MapReduce, running a background task that allows the Master to check if Workers are still alive

7 Mobile Code

So far, we looked at communication as **passing data**: Serialized procedure calls, resource representations, queries, or events

- Passing programs (**code migration**) can improve the design of distributed systems
- **Process migration** is also possible: Passing not only the code but also execution state

7.1 Reasons for migrating code

- **Performance**:
  - Distribute the load: Move tasks or jobs from heavily loaded components to lightly loaded components
  - Minimize network communication: Ensure computations are close to where the data is
- **Privacy and security**: Move the code to where the data is such that the data remains within the boundaries of some organization (ex: federated learning)
- **Flexibility**: Move the code to a component when needed (ex: a script for validating the input to an HTML form)

There are 3 styles of mobile code, depending on who initiates the code migration.

7.2 Remote Evaluation

Remote evaluation is a client-initiated code migration. The client has the know-how required to perform a service, but lacks the resources. Therefore, the client sends the code representing the know-how to the server, the server executes the code, and returns the result.

This simplifies the clients and allows for dynamic behaviour based on the sent data. However, the server must know and authenticate all clients to protect its resources.

7.3 Code-on-Demand

Code-on-Demand is a server-initiated code migration. The client has the resources required to perform a service, but lacks the know-how. The client requests the code representing the know-how from the server, receives that code, and executes it.

This increases the extensibility of clients and scalability of servers. Downloading code can also be done anonymously, since servers don't need to authenticate clients, but the clients have to trust the code.

7.4 Mobile Agents

Lastly, we have mobile agents, where both the client and server can initiate code migration. This combines the trade-offs from remote evaluation and code-on-demand. While this maximizes Flexibility, it is difficult to implement and less common in practice.

8 Distributed Computation: Spark

We have seen how MapReduce hides away many details regarding distribution transparency:

- Parallelization and distribution of application code
- Monitoring and balancing load over servers
- Dealing with failing machines
- Scheduling of input/output operations

However, MapReduce is not perfect:

- It has limited expressivity: Everything is written in either a map or a reduce function
- It is inefficient for iterative multi-pass algorithms, due to lots of disk access

Apache Spark is a generalization of MapReduce into data flows:

- Data flows can include multiple stages
- Data flows can use diverse transformations

8.1 Resilient Distributed Datasets (RDDs)

The main abstraction in Spark is that of a **Resilient Distributed Dataset (RDD)**:

- read-only collection of objects partitioned in memory or disk across a set of machines
- built via parallel transformations (map, filter, reduce, join, etc.)
- are resilient because it can be rebuilt if lost (fault-tolerance)
- can explicitly be cached in memory across machines to reuse it in multiple MapReduce-like parallel operations

With every operation, a new RDD is built, which encapsulates information about how it was derived. This creates a lineage graph and is useful to optimize computation and recover from failures by only recomputing lost partitions.

There are 2 types of operations in Spark:

- **Transformations**: Operations that take one or more RDDs and output a new RDD. There are 2 special transformations: `cache()` and `persist()`, which persists an RDD in the memory or disk of the workers. An RDD is persisted only when an action is invoked, which is useful to avoid redundant computations when multiple actions follow. RDDs with long lineages can be persisted to disk to optimize recovery from failure (like checkpoints).

- **Actions**: Operations that compute an RDD to return a value to the master, or to write data to external storage. Common actions include
  - `count()`: Returns the number of elements in an RDD
  - `collect()`: Returns the elements of the RDD
  - `saveAsTextFile()`: Saves the RDD to external storage as text file

RDD Operations can be lazy evaluated, which means transformations will only be calculated once an action is called.

8.2 Spark Execution in Four Steps

Apache Spark is a distributed computing system that processes large datasets in parallel. The core abstraction in Spark is the Resilient Distributed Dataset (RDD), which represents a fault-tolerant collection of elements that can be processed in parallel. When executing a Spark job, there are four main steps involved:

8.2.1 1. Create a DAG of RDDs to Represent the Computation

Directed Acyclic Graph (DAG) is a finite directed graph with no directed cycles. In the context of Spark, each RDD is represented as a node, and transformations (like `map`, `filter`, etc.) are represented as edges in this graph.

How it works:

- When a user submits Spark code, the system builds a DAG of RDDs. This graph captures the dependencies between RDDs, representing the sequence of computations.
- RDDs that need to be computed from data in storage are called “narrow dependencies” because they can be computed on a single partition of the parent RDD. Conversely, “wide dependencies” (like `groupByKey`) may require repartitioning because they are dependent on a number of partitions.

8.2.2 2. Create a Logical Execution Plan for the RDD Graph

Logical Execution Plan: Based on the DAG of RDDs, Spark builds a logical execution plan. This plan divides the DAG into stages, where each stage contains one or more tasks.

How it works:

- Stages are created by analyzing the DAG for transformations that have narrow dependencies. Each stage can be computed in parallel.

- Wide dependencies, such as those arising from `reduceByKey`, introduce boundaries that result in multiple stages.
- The result is a new DAG, but this time representing stages of tasks instead of RDD transformations.

8.2.3 3. Schedule Individual Tasks

Task Scheduling: Once the logical execution plan is created, the scheduler schedules tasks to Spark's worker nodes.

How it works:

- The Spark scheduler places tasks based on data locality, trying to run a task on a node where the data resides in memory or on-disk.
- It considers available resources and task dependencies. If an RDD's partition is available in memory on a node, the corresponding task will be scheduled on that node.
- The scheduler operates in a FIFO manner but can also be configured for other scheduling modes, such as fair scheduling.

8.2.4 4. Execute Individual Tasks

Task Execution: Tasks are the smallest unit of work in Spark, and they run on individual partitions of the RDD.

How it works:

- Worker nodes execute tasks that operate on local data. The result of each task is stored in memory, and if there's not enough memory, it spills to disk.
- Once a task is completed, the worker node reports back to the driver (the main Spark context), sending metadata and the location of the task's output.
- When all tasks in a stage are completed, the next stage's tasks are scheduled and executed.

Looking at how Spark executes operations, the importance of good partisioning is evident.

Having too few partitions is suboptimal

- Less concurrency than what could be achieved with the same computing power
- Large partitions increase memory pressure for operations like groupBy, reduceByKey, etc.

Having too many partitions is also suboptimal, due too much scheduling overhead

Partition tuning is the task of finding a reasonable number of partitions

- lower bound, ex: at least ~2x number of CPU cores in the cluster
- upper bound, ex: ensure tasks take at least 100ms

8.3 Conclusion

8.3.1 Spark's Innovations Over MapReduce

Spark has significantly enhanced both expressivity and performance over MapReduce, bringing forth several key innovations:

- **Diverse Transformations**: Unlike the limited scope of map and reduce in MapReduce, Spark supports a much broader set of transformations.
- **Multi-stage Processing**: Spark facilitates multi-stage data processing. Crucially, it enables in-memory caching between transformations, eliminating the need to always write intermediate results to external storage as is common in MapReduce.
- **Lineage Graphs**: These graphs empower Spark to optimize the execution flow of data. Furthermore, in the event of failures, Spark's lineage graphs allow for efficient recovery by recomputing only the lost partitions.

8.3.2 Batch Processing and Adoption

Similar to MapReduce, Spark's primary design goal was to cater to batch processing. Specifically, it aimed to handle the challenges of processing vast amounts of batch data, often termed as "Big Data". The success of this design philosophy is evident. Spark has not only been widely adopted but has also become a cornerstone in the Big Data processing world.

9 Distributed Storage: GFS

Distributed storage is another key layer of abstraction for distributed systems.

A couple of challenges include:

- Horizontal scalability: Distributing data among many servers (called sharding)
- Fault-tolerance: Replicating the data because faults are the rule rather than the exception
- Consistency: The replicated data should be consistent. Ideally, we want strong consistency, meaning all clients read the same data. The opposite, weak consistency, means depending on the server the data is read from, clients may receive different data.

9.1 Google File System (GFS)

The Google File System (GFS) is a distributed file system designed to provide fault tolerance, scalability, and reliability for large-scale data-intensive applications. It was developed by Google to support their internal infrastructure, and its design has significantly influenced the development of other distributed storage systems.

GFS hides away many details regarding distribution transparency:

- Sharding of files over many servers and disks
- Monitoring and balancing load over servers
- Dealing with failing machines

9.1.1 Key Design Principles and Assumptions

1. **Hardware Failures are Common**: Given the scale at which Google operates, it is expected that hardware components (such as disks and nodes) will fail on a regular basis. As such, GFS is built to handle these failures gracefully without significant disruptions.
2. **Files are Large**: Unlike traditional file systems that handle many small files, GFS is optimized for a small number of very large files.
3. **Mutations are Appends**: Files in GFS are primarily appended to rather than overwritten. This reduces the complexity of ensuring data consistency across replicas.
4. **High Throughput over Low Latency**: While GFS does aim to provide reasonable latency, its primary focus is on providing high throughput, especially for data-intensive operations.
5. **Application specific design**: Applications and GFS are co-designed and optimized to work together, which allows for flexible consistency.

9.1.2 Key Features

1. **Chunk-based Storage**: Files in GFS are divided into chunks, typically 64 MB in size. Each chunk is identified by a unique ID and is replicated across at least 3 nodes to ensure fault tolerance.
2. **Master and Chunk Servers**: GFS has a single master server responsible for metadata operations, and multiple chunk servers that store the actual file chunks. The master server periodically communicates with each chunk server to gather state information, ensure system health, and balance the load.

3. **Consistency and Coherency Model**: GFS provides a relaxed consistency model where, after a sequence of successful mutations, replicas are guaranteed to be eventually consistent. In case of inconsistencies, it uses a versioning mechanism to detect old replicas.
4. **Garbage Collection**: Instead of immediately deleting files, GFS marks them for deletion and periodically reclaims the space through a garbage collection process. This process is handeled by the master server.
5. **Snapshot Capability**: GFS can create quick, point-in-time snapshots of files or directories. This is done by copying metadata and not the actual data, making the process efficient.

9.1.3 Limitations

1. **Single Master Architecture**: Having a single master can become a bottleneck for metadata operations, especially in extremely large deployments. Therefore, clients read and write directly to chunkservers, and clients cache chunk metadata.
2. **Not Suited for Small Files**: Since GFS is optimized for large files, it can be inefficient when dealing with many small files.
3. **Manual Intervention**: Some maintenance tasks, especially in the early versions of GFS, required manual intervention.
4. **Expensive Write Operations**: A write operation to a chunk is performed on all the chunk's replicas. To handle parallel writes, write operations are serialized and applied in the same order by all replicas. This is called coordination and is handeled by a primary replica selected by ther master.

9.1.4 Successors and Influence

The design principles and lessons learned from GFS paved the way for other distributed storage systems. One notable successor is the **Hadoop Distributed File System (HDFS)**, which is heavily influenced by GFS. HDFS is used in conjunction with the Hadoop ecosystem for distributed data processing tasks and provides better consistency gaurantees than GFS.

10 Consistency and Logical Time

Consistency is a very important topic in distributed systems to optimize performance and ensure accurate data manipulation.

In the context of this chapter, any time difference is in real-time (i.e. wall-time).

10.1 Data-Centric Consistency Models

In the context of data centers, the consistency gaurantees should be strong enough to be useful for the user, but weak enough to be easily implemented. That means there is a trade-off between convenience and speed.

There are a couple of important consistency models (consistency decreases):

1. Linearizability
2. Sequential consistency
3. Casual consistency
4. Eventual consistency

The best model is highly dependent on the application.

To model consistency models, we use a special notation:

- **Object**: A variable or data structure storing information
- **Operation** f: Accesses or manipulates an object. The operation f starts at wall-clock time f_s and completes at wall-clock time f_c
- **Execution** E: A set of operations on one or multiple objects that are executed by a set of nodes.

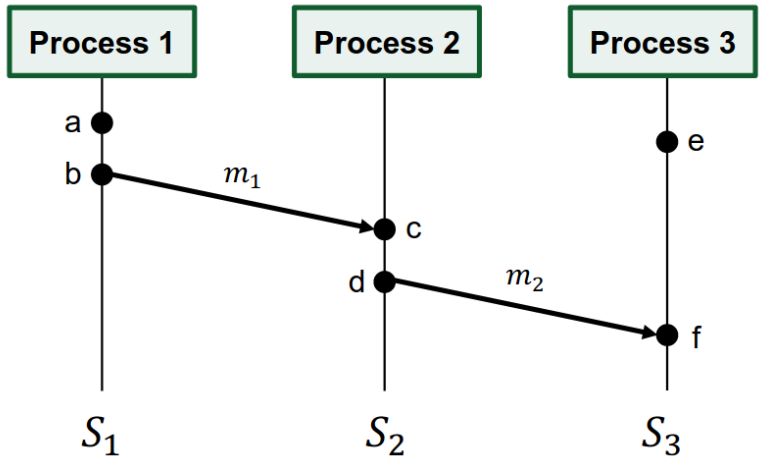
If f and g are two distinct operations, where $f_c < g_s$, we say f occurs before g and write $f < g$.

Let S_u be a sequence of operations on some node u and define $\rightarrow$ to be the **happened-before relation** on $E = S_1 \cup \cdots \cup S_n$ that satisfies the following three conditions:

1. If a local operation f occurs before operation g on the same node ($f < g$), then $f \rightarrow g$.
2. If f is a send operation of one node, and g is the corresponding receive operationn of another node, then $f \rightarrow g$.
3. If f, g, h are operations such that $f \rightarrow g$ and $g \rightarrow h$, then also $f \rightarrow h$.

If for two distinct operations f, g neither $f \rightarrow g$ nor $g \rightarrow h$, then we say f and g are concurrent and write $f||g$.

For example:



- $a \to b, c \to d, e \to f$ due to node execution order
- $b \to c$ and $d \to f$ due to messages m_1 and m_2
- $a \to c, a \to d, a \to f, b \to d, b \to f, c \to f$ due to transitivity
- $a||e, b||e, c||e, d||e$

In the examples, we use a shortened notation for read and write operations:

- "Write value 1 to x" $\to Wx1$
- "Read value 2 from x" $\to Rx2$

10.1.1 Linearizability

Linearizability is the strongest consistency model, simulating a system where all operations occur instantaneously.

It requires:

- Each operation f to appear instantaneously at some point between its start f_s and completion f_c.
- All operations to act as if executed on a single copy of the data.

For overlapping operations, we choose a linearization point $f_* \in [f_s, f_c]$ for each operation f, at which the operation is executed. An execution E is linearizable if and only if there exists linearization points such that the sequential execution S that results in order the operations according to those linearization points is semantically equivalent to E.

Key Points:

- **Real-Time Ordering:** Operations are ordered as if they are happening in real-time, even if they occur concurrently.

- **Global State Visibility:** Every operation is immediately visible to all nodes in the system once completed.
- **Predictability:** Ensures a predictable, consistent state across all nodes at any given time.
- **Concurrency Handling:** Overlapping operations are ordered in a way that respects the real-time occurrence of events.

Example:

Process P1	Process P2
Write x = 1	
	Read x = 1
Read x = 1	

In this example, P2's read operation occurs only after P1's write operation has completed, reflecting linearizability's requirement for real-time consistency.

Applications:

- Financial systems where transactions must be reflected in real-time.
- Systems requiring immediate consistency, like online booking platforms.

10.1.2 Sequential Consistency

Sequential consistency is less strict than linearizability and focuses on preserving the order of operations as issued by each node.

It requires:

- An execution E to have a sequence S that is equivalent to E in terms of operations and their outcomes.
- Operations from a single node to appear in order in S as they were issued in E.

Key Points:

- **Node-based Ordering:** Maintains the order of operations as they are issued by individual nodes, but not necessarily in real-time.
- **Independence from Global Time:** Operations across different nodes can be viewed in different orders, as long as the order per node is maintained.
- **Flexibility in Overlapping Operations:** Provides flexibility in how operations from different nodes are viewed in relation to each other.

Example:

Process P1	Process P2
Write x = 1	
Write x = 2	
	Read x = 2
	Read x = 1

Here, P2's reads can reflect the writes in a different order, which wouldn't be allowed under linearizability but is acceptable in sequential consistency.

Applications:

- Collaborative applications like document or code editing.
- Applications where the relative order of operations within the same node is more important than the absolute order across the system.

10.1.3 Causal Consistency

Causal consistency maintains the causal relationships between operations, ensuring that causally related operations are seen in the same order by all nodes.

It requires:

- An execution E to have a corresponding sequence S that mirrors E in operations and outcomes.
- If operation f causally precedes g in E (denoted $f \rightarrow g$), then f must also precede g in S.

Key Points:

- **Causal Relationships:** Preserves the order of operations that are causally linked.
- **Concurrent Operations:** Allows for different perceptions of the order of operations that are not causally related.
- **Flexibility:** More flexible than linearizability or sequential consistency, making it suitable for distributed systems where exact ordering of all operations is not critical.

Example:

Process P1	Process P2	Process P3	Process P4
Write x = 1			Read x = 1
	Read x = 1		
	Write x = 2		Write x = 3
Read x = 3		Read x = 2	

In this scenario, P3 reads the value of x as 2 after P2's write, which was causally linked to P1's write. This sequence respects causal relationships, allowed in causal consistency but not necessarily in sequential consistency.

Applications:

- Social media platforms where the order of comments and reactions to a post is important.
- Real-time collaborative tools, like whiteboards or collaborative editing, where the context of changes matters.

10.1.4 Eventual Consistency

Eventual consistency is the most relaxed model, ensuring that all nodes will eventually have the same data if no new updates are made.

It requires:

- If no new updates are made to a given data item, all accesses to that item will eventually return the last updated value.

Key Points:

- **Eventual Data Convergence:** All nodes will eventually hold the same data, but not necessarily at the same time.
- **Tolerance to Inconsistencies:** Temporarily tolerates inconsistencies for better system availability and partition tolerance.
- **Scalability:** Highly scalable, as it does not require immediate consistency across all nodes.

Example:

Process P1	Process P2
Write x = 1	
	Read x = 0
	Read x = 1

P2 initially reads an old value of x, but eventually reads the updated value. This temporary inconsistency is acceptable under eventual consistency, unlike in the stricter models.

Applications:

- Large-scale distributed databases where immediate consistency is less critical than availability.
- Content delivery networks (CDNs) and caching systems where data can be slightly out-of-date but highly available.

10.2 Logical Time

Logical time systems, such as Lamport's Logical Clocks and Vector Clocks, play a crucial role in managing consistency in distributed systems, especially in the absence of synchronized physical clocks.

10.2.1 Strong Logical Clocks

Strong logical clocks extend beyond simple timestamping to enforce a stronger condition: if one event causally affects another (A → B), then the timestamp of A must be less than the timestamp of B (TS(A) < TS(B)). This is more than just recording the order of events; it ensures that the causal relationships are accurately reflected in the timestamps.

10.2.2 Lamport's Logical Clocks

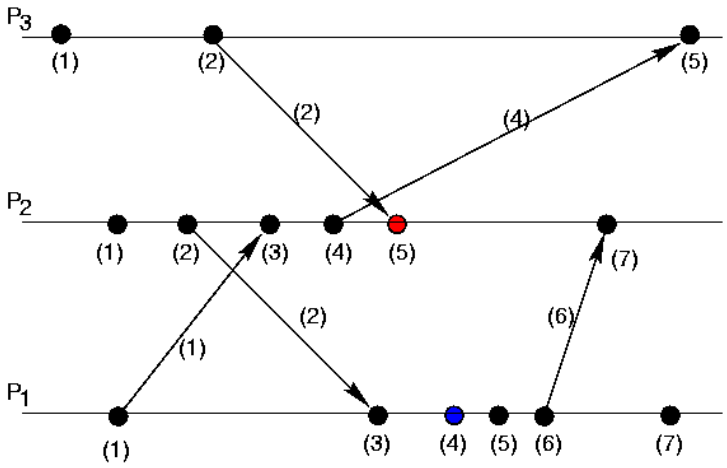
Lamport's Logical Clocks provide a way to order events in a distributed system using "logical time" instead of real time. They are based on the concept of a counter, which is incremented following specific rules:

1. **Each process maintains its own logical clock**, which is a counter.
2. **On executing an internal event**, a process increments its logical clock.
3. **For every message sent**, a process increments its logical clock and includes the current value of its clock with the message.

4. **On receiving a message**, a process sets its clock to be greater than the maximum of its current value and the value received in the message.

This system ensures that the “happens-before” relationship is maintained. If event A happens before event B in the system (denoted as A → B), then the timestamp of A is less than the timestamp of B (TS(A) < TS(B)).

Lamport’s logical clocks are not strong logical clocks because they only ensure a partial ordering of events based on the “happens-before” relationship. We need more information about other the processes to make it strong.



10.2.3 Vector Clocks

Vector Clocks are an extension of Lamport Clocks that provide more detailed information about the causal relationships between events. Each process in the system maintains a vector, with one entry for each process:

1. **Initial State**: Every entry in the vector is initialized to zero.
2. **On executing an internal event**, a process increments the value in its position in its vector clock.
3. **On sending a message**, a process increments its position in its vector clock and then sends the message with its entire vector clock.
4. **On receiving a message**, a process increments its position in its vector clock and updates each element in its vector clock to be the maximum of the value in its own vector clock and the value in the vector clock received with the message.

This mechanism allows the system to differentiate not only the order of events but also whether two events are causally related or concurrent. Vector Clocks are therefore strong logical clocks as they

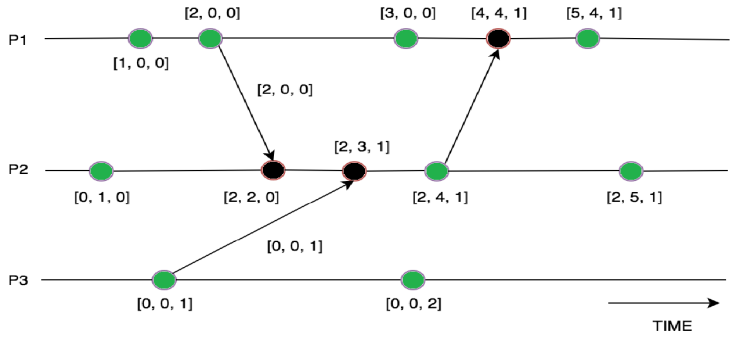
enforce a strict ordering.

Key Advantages:

- **Causal Relationship Awareness:** Vector clocks can distinguish between causally related events and concurrent events.
- **Better Insight into System State:** They provide a clearer picture of the state of the system at any given time.
- **Conflict Resolution:** Useful in resolving conflicts in distributed data stores, as they can help identify if one operation causally follows another.

Applications:

- **Distributed Databases:** For resolving conflicts in replicated databases.
- **Distributed Version Control Systems:** Like Git, where understanding the order and causality of changes is crucial.
- **Real-time Collaborative Tools:** Where it's essential to understand the sequence of user actions.



11 Fault tolerance

When dealing with distributed systems, server failures are very common. There are many different causes for failures:

- Crash failure: server halts prematurely, but was working correctly until it stopped
- Omission failure: server fails to respond to request
- Timing failure: the server's response lies outside a specified time interval
- Response failure: the server's response is incorrect
- Arbitrary (Byzantine) failure: may produce arbitrary responses at arbitrary times. This also means there is inperfect information if a component has actually failed

Dealing with faults depends on the timing behaviour of the distributed system. There are 3 key behaviours:

- Synchronous system: both process execution speeds and message-delivery times are bounded. In practice, this is usually not the case (network latency or execution speed can vary wildly)
- Asynchronous system: no assumptions about process execution speeds or message delivery can be made (messages can be delayed arbitrarily, nodes can pause execution, etc.)
- Partially synchronous system: the system behaves as a synchronous system most of the time, but there is no bound on the time it behaves in an asynchronous fashion.

In the following sections, we will consider a partially synchronous system, as it is the most common. In such a system, we can use heartbeats and timeouts to detect crashes. However, there might be wrong conclusions, for example, if a server's computation takes longer than the defined timeout.

We will now investigate ways to hide process failues thorugh redundency.

11.1 Primary-Backup Protocols

In primary-backup protocols, each data object has a number of replicas with an assigned primary replica that handles write operations. Specifically, all write operations are forwarded to the primary replica, while read operations are handled by the replica that received the request. The primary is usually fixed, but can be replaced if needed, for example in case of a crash of the primary replica. In that case, the backup replicas can execute a selection algorithm to determine the new replica.

This provides a straight-forward way for sequential consistency as there is only a single process, the primary replica, that makes the decision. However, a write operation is expensive, as it has to be replicated across all replicas and blocks other write operations in that time.

11.2 Qourom-Based Protocols

Now let's assume we have a group of servers with no assigned roles. This removes the single point of failure, but creates a new issue: How do we achieve consensus?

In a read-after-write consistent protocol, the client always reads the last value it has written or a more recent one. If the client receives different responses from servers, the client has to choose, which value to accept. This is usually done through timestamps. The set of all replicas that reply is called the qourom.

We can calculate how many replicas can fail before we loose consistency. Let's assume we have n replicas and the client creates a read qourom with r replicas (these are the replicas requested for reads) and w replicas for a write qourom, then:

- if $r + w > n$, there are no read-write conflicts (the system maintains read-after-write consistency)
- If $w > \frac{n}{2}$, there are no write-write conflicts because the majority of nodes have the latest writes.

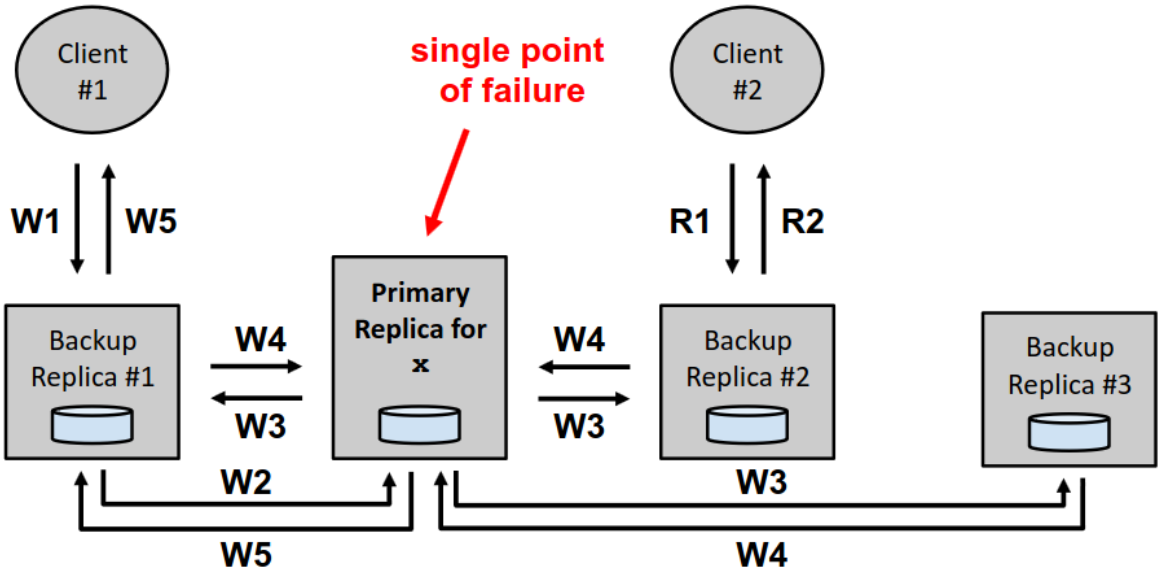


Figure 3: Primary-Backup Protocols

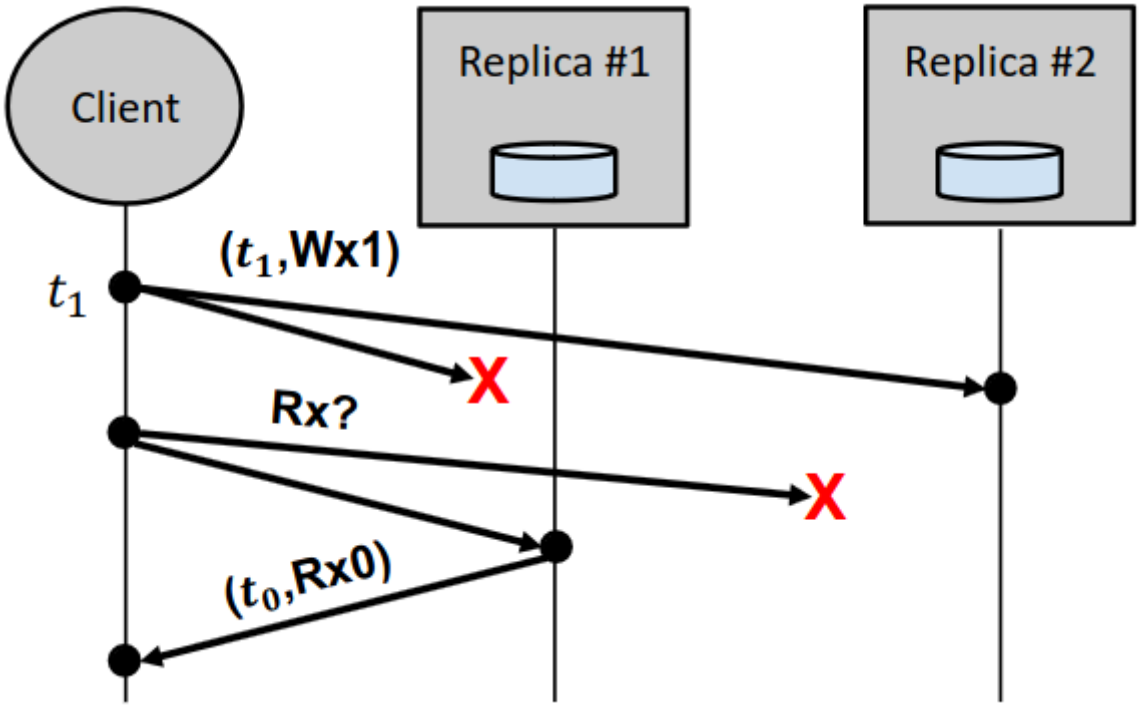


Figure 4: Read-after-write consistency - Client sees (t_0,Rx0) — instead of (t_1,Rx1)

A system is k-fault tolerant if it can withstand k failing processes. If k processes fail silently (fail without causing any further complications), then only $k + 1$ processes are needed to keep the system operational. However, if k processes fail arbitrarily (exhibit any kind of incorrect behavior, including providing wrong or misleading responses), then $2k + 1$ processes are needed to keep the system operational (the working processes form the majority).

As a qourom grows, so does the cost of performing an operation. This means that by tweaking r and w, we can prioritize availability (for example response time) vs. consistency.

11.3 Raft Consesus Algorithm

Raft decomposes the consesus problem into 3 sub-problems:

1. Leader election: Select one node to act as leader. When the leader fails, select a new leader
2. Log replication: The leader accepts operations from client, appends them to its log, and replicated the log to the other nodes. This overwrite any inconsistencies
3. Safety: Keep logs consistent despite failures. Only nodes with up-to-date logs can be elected as leaders

Raft is not a Byzantine fault tolerant algorithm as the nodes trust the elected leader. It provides high availability, but all nodes are doing the same task.

Every server in a raft cluster can be a leader or a follower. In case of new leader eledction, a server can also assume the role of candidate (if it thinks the leader is offline).

A raft term starts with this new election. If the election is succesful, the term keeps going with normal operations orchastrated by the leader. If the election fails, a new term starts. The current term is sent with every RPC and is checked by the receiving server.

11.3.1 Leader Election

A leader election is started by a candidate server. It starts the election by increasing the term counter, voting for itself as new leader, and sending a message to all other servers requesting their vote. A server will vote only once per term, on a first-come-first-served basis. If a candidate receives a message from another server with a term number larger than the candidate's current term, then the candidate's election is defeated and the candidate changes into a follower and recognizes the leader as legitimate. If a candidate receives a majority of votes, then it becomes the new leader. If neither happens, e.g., because of a split vote, then a new term starts, and a new election begins. A split vote is resolved quickly by using an election timeout. This should reduce the chance of a split vote because servers won't

become candidates at the same time: a single server will time out, win the election, then become leader and send heartbeat messages to other servers before any of the followers can become candidates.

Every candidate includes the index and term of their last log entry when requesting votes. Other nodes deny their votes if their log is more up-to-date. This ensures that only servers with up-to-date logs can become leaders.

11.3.2 Log Replication

Clients always send operations to the current leader (if a client contacts a follower, the client is redirected to the leader). The leader responds to the client, appends the operations to its log, and sends RPCs to all followers.

Logs are persisted on disk to survive crashes. An enty is commited only if it is safe (replicated on a majority of nodes). During a commit, the leader executed the operations and returns the result to the client. The leader then notifies followers of commited entries in subsequent RPCs. Follower execute the operations themselfs.

The leader assumes its log is correct and overwrites inconsistencies with these steps:

- RPCs include the index and term of the log entry preceding the new entry
- A follower's log must contain matching entry to append the new entries - or otherwise it rejects the request
- If the request is rejected, the leader tries again with a lower log index

This way, the leader and follower can find the last matching entry and overwrite from there.

12 Distributed Transactions

12.1 ACID Transactions

In a distributed environment, ACID (Atomicity, Consistency, Isolation, and Durability) transactions are crucial for maintaining data integrity and ensuring reliability. These properties are defined as:

- **Atomicity**: A transaction is an all-or-nothing unit of work. Either all operations in the transaction are completed successfully, or none are applied.
- **Consistency**: Transactions transform the system from one consistent state to another. Any data written will be valid according to all defined rules, including constraints, cascades, and triggers.
- **Isolation**: Transactions are isolated from each other. The intermediate state of an ongoing transaction is invisible to other transactions. There are various levels of isolation that trade-off between performance and the degree of visibility of intermediate states.

- **Durability**: Once a transaction is committed, it will remain so, even in the event of power loss, crashes, or errors. Typically, this is achieved by ensuring that the completed transactions are recorded in a non-volatile memory.

To allow for ACID transactions while allowing concurrency is often implemented through **lock-based protocols**. A lock can be

- Exclusive (write lock): The object can be both read and written by a single client
- Shared (read lock): The object can only be read by multiple clients

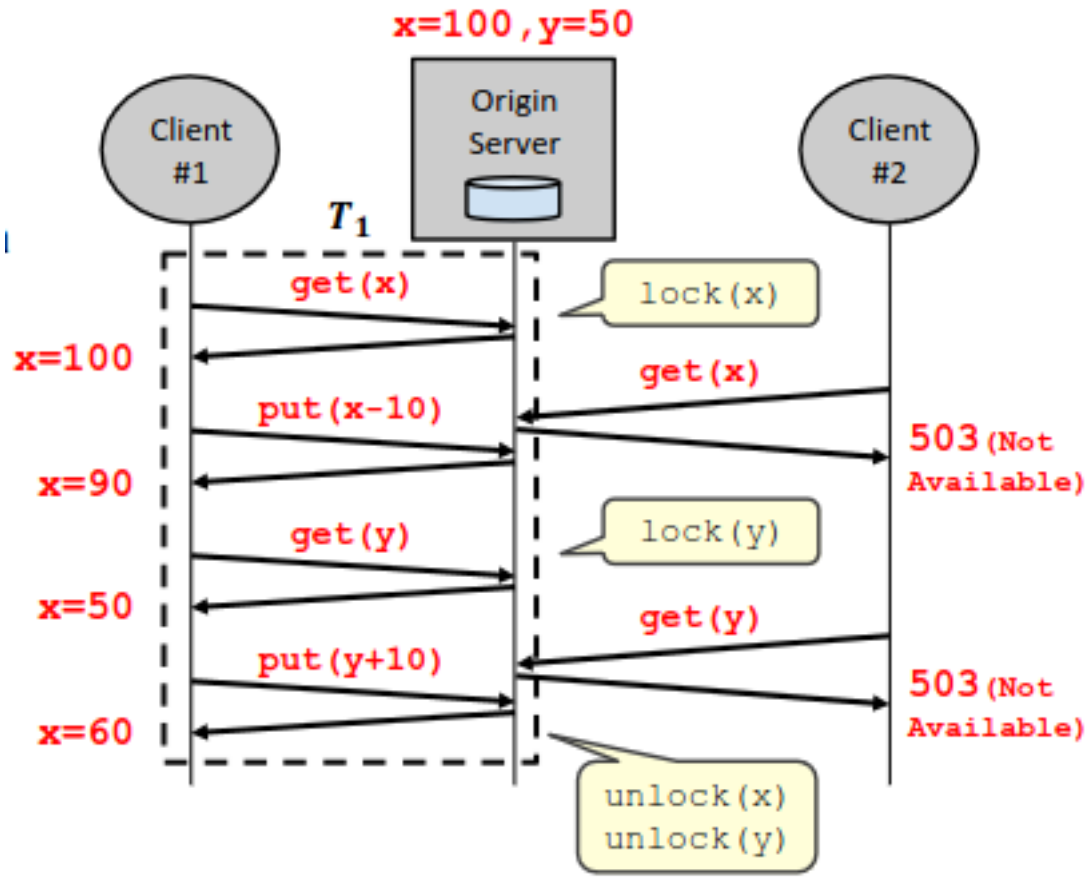


Figure 5: ACID Transactions Properties

These properties ensure a predictable, reliable behavior for complex operations in a distributed system.

12.2 Two-Phase Commit Protocol

The Two-Phase Commit (2PC) Protocol is a type of consensus protocol used to ensure that all participating nodes in a distributed system agree on a proposed transaction. It is an algorithm used for maintaining the atomicity and consistency aspects of ACID in distributed transactions. The protocol works in two phases:

1. **Voting Phase (Phase 1)**: The coordinator proposes a transaction and asks all participants (nodes) to vote. Each participant will vote 'Yes' if it is ready to commit the transaction or 'No' if it cannot. All participants will then hold the resources and lock the necessary data, waiting for further instruction.
2. **Commit Phase (Phase 2)**: Depending on the votes collected in the first phase, the coordinator decides to either commit or abort the transaction. If all participants voted 'Yes', the coordinator sends a commit message. Each participant then commits the transaction and releases all locks and resources. If any participant voted 'No', the coordinator sends an abort message, and each participant aborts the transaction and releases all locks and resources.

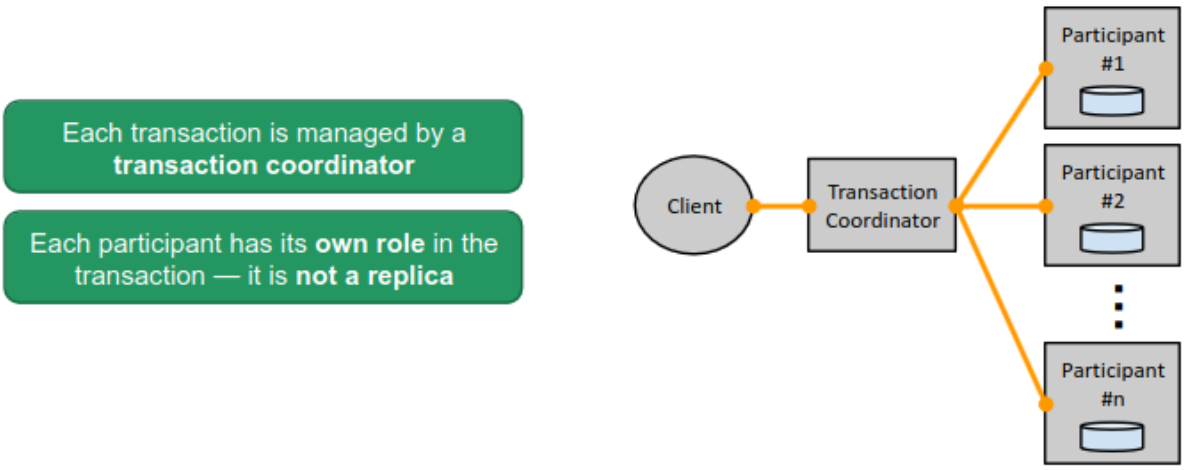


Figure 6: Two-Phase Commit Protocol

The 2PC ensures consistency and atomicity in a distributed system but has some drawbacks, such as being a blocking protocol (if the coordinator fails, participants remain in an uncertain state) and requiring all participants to be available during the voting phase. Despite its shortcomings, 2PC is widely used in distributed databases to maintain transaction integrity.

To recover from failures, participants must persist transactions, their state (init, waiting for coordinator, commit, abort), and the coordinator's decision. This allows the participant to recover at any state in case of a crash. If a participant is waiting for the coordinator's decision, crashes, and does not have the decision persisted, then it can also ask the coordinator for the decision.

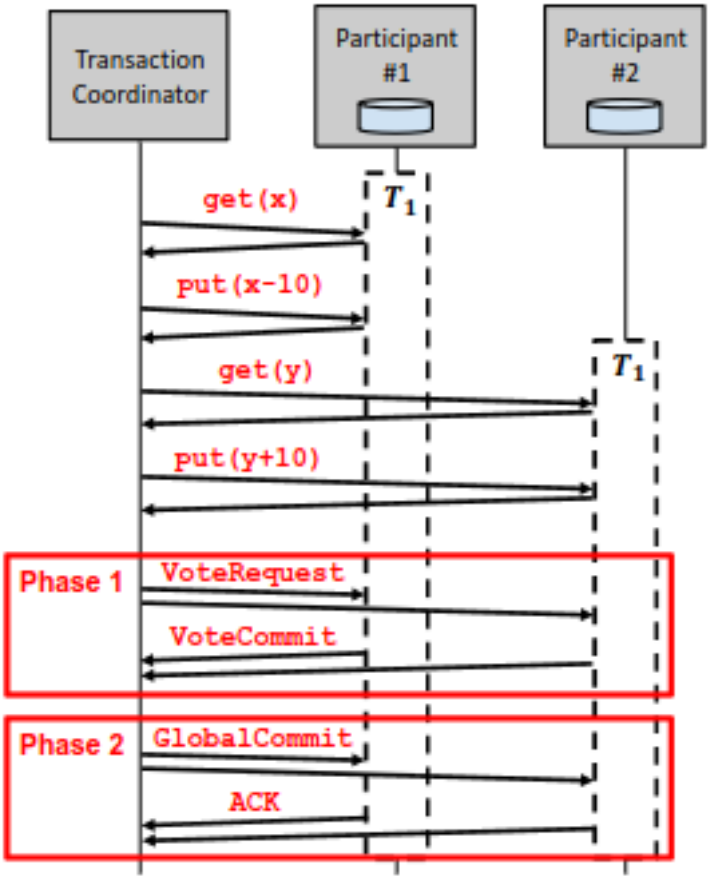


Figure 7: Two-Phase Commit Example

The coordinator also must save its state to persistent storage for recovery. However, if the coordinator fails before sending the final decision, participants are blocked.

2PC ensures atomic commits but it is slow. However, each node can handle a different part of the transaction.

12.3 2PC over Raft

We can use consensus algorithms such as Raft to replicate both the coordinator and the participants. This allows for both high availability and atomic commits.

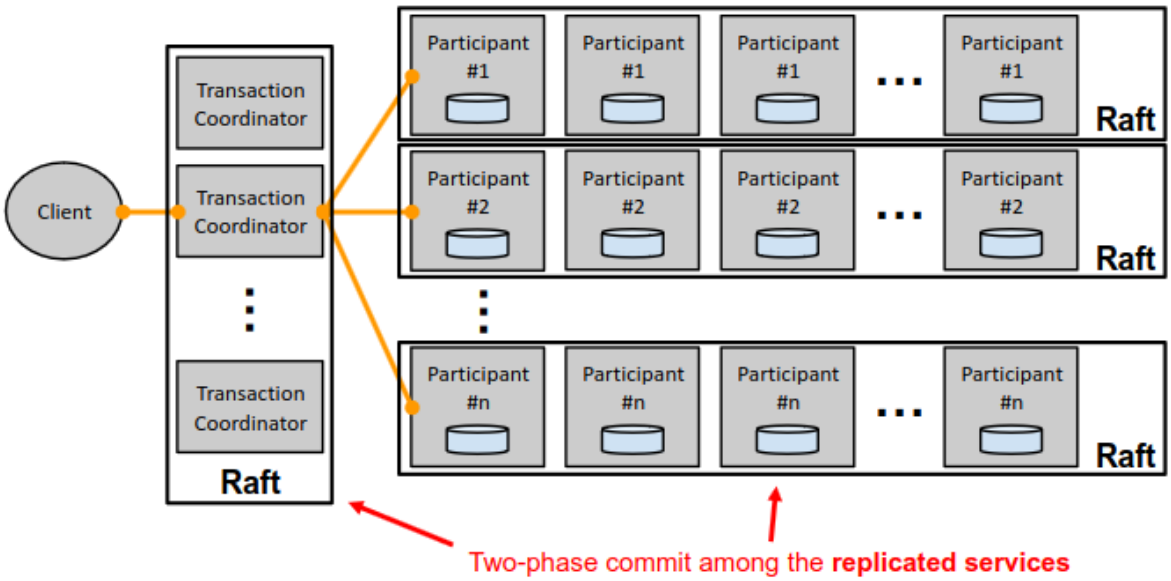


Figure 8: 2PC over Raft

Google uses a similar technology for their datacenters, called Spanner. It uses 2PC over Paxos (similar to Raft) and allows for millions of machines and trillions of database rows. It allows for schematized tables, SQL-like queries, ACID transactions, and global distribution.

In Spanner, tables are partitioned (sharded) horizontally into tablets and distributed across machines. The sharding is transparent to clients. It uses TrueTime, which allows Spanner to achieve external consistency (a stronger form of consistency than linearizability) for globally distributed transactions. By leveraging precise time bounds, Spanner can order transactions correctly across the entire system. Specifically, it calculates two points in time, one in the past and one in the future on every server, and can wait out the uncertainty created by these bounds.

Does Spanner break the CAP Theorem? No, Spanner seems to tick all three boxes (consistency, availability, partition-tolerance), it favors consistency over availability. Therefore, it is a CP system.

13 Dezentralization and Blockchain

Bitcoin addresses the complex issue of reaching consensus in the presence of Byzantine failures, a situation where components may fail and there's imperfect information about whether a component has failed. The system incorporates two key innovations:

1. **Public Ledgers Using Cryptographic Primitives**: Bitcoin utilizes a public ledger, commonly known as the blockchain, to record all transactions. This ledger is maintained through cryptographic techniques, ensuring that once a transaction is recorded, it cannot be altered retroactively without altering all subsequent blocks. This feature provides a high level of security and transparency. The Bitcoin blockchain uses an input hash for every block to calculate a new output hash, which is then used as input for the next block.
2. **Leader Election Through Proof-of-Work**: To add a new block of transactions to the blockchain, a process called "mining" is employed. This involves solving a computationally intensive problem, known as proof-of-work, where a hash of a block with N zeros is calculated. The first miner to solve the problem gets to add the new block to the blockchain and is rewarded with new bitcoins. This method is highly resistant to Sybil attacks, where an attacker might create a large number of pseudonymous identities to gain a disproportionately large influence.

However, Bitcoin's design presents several challenges:

- **Fixed Block Size and Mining Frequency**: Each block in the Bitcoin blockchain has a fixed size of 1 MB, and blocks are mined approximately every 10 minutes. This structure limits the maximum transaction throughput, acting as a bottleneck in the system and potentially leading to delays during times of high transaction volumes.
- **Transaction Confirmation Time**: Transactions in Bitcoin require multiple confirmations for validation. Each confirmation corresponds to a new block being added to the chain, which can take significant time, causing delays in transaction finalization. Confirmation works by waiting for temporary forks to be discarded by a majority of nodes, who decide to work on the longest ("honest") chain.
- **System Flooding**: Flooding the network with new blocks can limit overall performance, as it leads to increased data to process and verify.
- **Energy Consumption of Proof-of-Work**: The proof-of-work mechanism, while effective against certain types of attacks, is energy-intensive and has been criticized for its environmental impact.
- **Vulnerability to Majority Attacks**: If a majority of the network's computing power is controlled by a single entity or a colluding group, they could potentially manipulate the blockchain, though this remains a theoretical risk rather than a common occurrence.

- **Centralization through Mining Pools**: The emergence of mining pools, where miners combine their computational resources to increase their chances of successfully mining a block and receiving rewards, has led to concerns about centralization. This centralization could potentially contradict the decentralized ethos of Bitcoin and might pose risks to its security and integrity.

14 Distributed Intelligence

14.1 Autonomous Agents and Multi-Agent Systems

Distributed Intelligence encompasses the concept of autonomous agents and multi-agent systems (MAS). An **autonomous agent** is an entity capable of observing its environment through sensors and interacting with it through actuators. These agents can be either software, robots, or humans. Their autonomy allows them to perform tasks without continuous guidance from a user or another system.

14.1.1 Characteristics of Multi-Agent Systems (MAS)

A **multi-agent system** consists of multiple autonomous agents interacting within a shared environment. These agents can operate under various modes:

- **Collaboration**: Agents work towards shared goals.
- **Cooperation**: Agents work towards compatible but distinct goals.
- **Competition**: Agents work towards incompatible goals.

MASs exhibit several key traits:

- **Decentralized Control**: No single agent has complete control over the system.
- **Distributed Capabilities**: Knowledge and resources are spread across different agents.
- **Inter-agent Dependencies and Conflicts**: Agents must coordinate and may have conflicting interests.
- **Coordinated Behavior**: The overall system demonstrates cohesive behavior despite the decentralized nature of control.

These characteristics make MAS particularly challenging and interesting, as they mirror many aspects of distributed systems but with added complexities due to the autonomous nature of the agents involved.

14.1.2 The Semantic Web and Multi-Agent Systems

The concept of the **Semantic Web** is integral to distributed intelligence. It involves using ontologies, knowledge graphs, and linked data to create a web of data that can be easily interpreted and utilized by machines, including autonomous agents. This web of data provides a rich, interconnected environment for agents to interact with and derive meaningful insights.

- **Ontologies**: Formal, explicit specifications of shared conceptualizations, providing a structured framework for information.
- **Knowledge Graphs**: Graphs of data intended to accumulate and convey real-world knowledge, with nodes representing entities and edges representing relationships.
- **Linked Data**: Connecting data across different sources to enable more comprehensive understanding and insight.

By integrating these concepts, multi-agent systems can leverage the vast and interconnected data available on the Semantic Web, allowing for more intelligent and context-aware decisions.

14.2 Solid

Solid, is a standard for allowing users to store their data in Personal Online Data stores (PODs), which they control. It is developed by WWWW inventor Sir Tim Berners-Lee to enhance data privacy and autonomy.

The idea is to decouple data storage and services. The data stored in PODs can be selectively shared with services. These permissions can be very granular and revoked at any time.