Jacob Chmura

Introduction to Distributed Systems

Introduction

Definition 1. A **distributed system** is a collection of *autonomous computing elements* or **nodes** that appear to its users as a single coherent system

- Nodes must be able to act independently of each other, and must be coordinated and synchornized via message passing
- Group membership requires a mechanism to establish new nodes, evict old ones, and manage the set of nodes a given computing element can directly communicate with
- Nodes are organized into an **overlay network**. In a *structured overlay* each node has a well-defined set of neighbors with whom it can communicate. In a *unstructured overlay*, each node has a number of references to randomly selected other nodes.

Definition 2. A **middleware** is a software application layer that assists the development of distributed systems by providing a unified interface between several networked nodes.

- *Communication*: A **remote procedure call RPC** service allows an application to invoke a function is implemented and executed on a remote computer as if it was locally available
- *Transactions*: An **atomic transaction** ensures execution of operations in an *all-or-nothing* fashion

Design Goals

- Support resource sharing (data, networks, processes, cpu, gpu)
- Making distribution transparent (access, location, failure, replication, concurrency)
- Being open (interoperability, portability, extensibility)
- Being scalable (size, geographical, administrative)
  - **scaling up** involves increasing memory, upgrading CPU, etc. There is an upper bound on this.
  - **scaling out** expanded distributed system by deploying more machines
    - asynchornous communication
    - partition and distribution
    - replication
    - caching

Types of Distributed Systems

- High performance Distributed Computing (cluster computing, grid computing, cloud computing)
- Distributed Information Systems
- Pervasive Systems (ubiquitous computing systems, mobile systems, sensor networks)

Definition 3. A **transaction** follows the **ACID properties**:

- *Atomic*: the transaction is all-or-nothing, it either fully completes, or doesnt change at all
- *Consistent*: the transaction does not violate system invariants
- *Isolated*: concurrent transactions do not interfere with each other
- *Durable*: once a transaction commits, its changes are permanent

Architectures

Definition 4. A **layered architecture** involves linearly seperating software components and making *downcalls* downs the hierachy

Definition 5. A **object based architecture** puts a remote object in place of a software component, and connects them through a procedure call mechanism on the object.

Definition 6. A **resource based architecture** like **Representational state transfer (REST)** involve *stateless execution* that implements (PUT, GET, DELETE, POST) interfaces on a resource based naming scheme.

Definition 7. A **publish subscribe architecture** decouples processing and coordination via event based coordination. A process can *publish* the occurrence of an event, and other process may *subscribe* to specific kinds of notifications.

There are two important design patterns that target the organization of middleware: wrappers and interceptors.

Definition 8. A **wrapper or adapter** is a component that offers an interface acceptable oto a client application of hwich teh functions are transformered into those available at the component (solving incompatible interfaces).

A **broker** is a centralized component that handles all accesses between application.

Definition 9. An **interceptor** is a software construct that breaks flow of control to allow other code to be executed.

System Architecture

A **client server architecture** has a *server* that implements a specific service and a *client* that request service from the server, waiting for a reply.

A **peer-to-peer** system is decentralized architecture where nodes are symmetrically and equally responsible for client and server activities.

Processes

Threads provide an efficient abstraction level the unlocks performance for distributed systems, bypassing communication and startup overhead of multi-processing.

- A *process context* is created by the OS when launching a new process, containing file handles, memory maps, static data, program counter, memory, registers etc.
- A consequence of transperency of execution is the requirement to create a completely new address space on every new process
- Changing execution of control also requires invaliding address translaton caches (TLB), and potentially swapping memory from ram to disc
- In a single threaded application, blocking calls suspend execution control from the process.

- With multi-processing, communication IPC mechanisms require kernel intervention which is slow
- **Many-to-one threading** creates threads in user space. They are low-overhead, fast switching, fast communication. The problem is blocking on one thread suspends the whole process so for example, thread-speific IO cannot be performed.
- **One-to-one threading** creates a new thread aware by the os. This solves the above problem, but adds overhead to every thread operation due to sys calls.
- Multi-threading is particularly used in distributed systems on the server side. One thread, the *dispatcher* reads incoming requests, and leveraging a set of *worker threads*, which block, but allow other workers to take over on subsequent requests.

Virtualization

Computer systems generally offer four different types of interfaces at three different levels of abstraction:

1. **Instruction set architecture** providing interface between hardware and software
   - Privileged instructions
   - General instructions
2. **System calls** offered by the operating system
3. **Application programming interface** providing library calls

Communication

Definition 10. 7 layer OSI Model

1. Physical layer: standardized bit encodings and architectures between computers
2. Data link layer: detects and correct transmission errors
3. Network layer: routing message through network, and handling congestion
4. Transport layer: establish reliable communication, support real-time streaming
5. Session layer: support for session between applications
6. Presentation layer: prescribe how data is represetned independent of hosts
7. Application layer: everything else

Definition 11. Connection-oriented service establish a connection and negotiate protocal parameters prior to exchanges data between sending and receiving process.

Definition 12. Connectionless services no setup in advance is needed.

Definition 13. Persistent communication involves having the middleware store the submitted message as long as it takes to deliver to the receiver.

Definition 14. Transient communication has a message be stored by communication system only as long as the sending and receiving application are executing.

Definition 15. Asynchronous communication, the sender continues immediately after it has submitted its message for transmission.

Definition 16. Synchronous communication the sender is blocked until its requires is accepted.

Remote Procedure Call

Idea: allow programs to call procedures implemented on other machines.

- A *client stub* is locally defined procedure header that is referenced by the caller
- The client stub packs the parameters into message (*marshalling*) and blocks waiting for remote call to return result from *server stub*
- Special care needs to be taken to choose a message encoding format
- Passing reference remotely doesn't make sense since they refer to local address space. The entire object must be copied over at some point

Message oriented communication

Definition 17. A **socket** is a communication endpoint to which an application can write data to be sent over the underlying network (TCP socket is not the same as ZMQ socket, ex. TCP is 1-1, but ZMQ supports many-to-one and multicasts).

The *socket address* consists of a host, and port combination.

- ZMQ is fully asynchronous and connection oriented: message queue on sending end, so that we can send without server being up and running
- ZMQ allows multiple transmission mechanisms (TCP, UDM, IPC, intra-process)
- ZMQ pattern sockets for re-use (REQ-REP, PUB-SUB, PUSH-PULL)

Definition 18. Message Queueing Systems are middleware that provide intermediate term storage and implement asynchronous communication between nodes through queue systems.

Definition 19. Message brokers are special nodes inside a queueing system that acts as a centralized application gateway for converting incoming messages to the appropriate format for the receiver queues, routing etc.

Coordination

Clock Synchronization

A **computer clock** is a timer that oscilates at a well-defined frequency, and broadcasts signal when register counter hits zero.

Internal synchronization involves keeping the clocks precise (consistent within groups of clocks). *External synchronization* involves keeping the clocks accurate (consistent with UTC).

Berkeley Algorithm (internal clock sync): time server daemon polls processes for their time, averages, and sends back delta times that should adjust each clock.

Lamport Logical Clocks

The *order in which events occur* is more important than absolute time.

Happens-before(a, b) = event a happens before event b We can directly observe this if:

- a, b in same process and a cocurs before b then $a \to b$ is true
- a is event message sent by one proces, b is message received by another, then $a \to b$ is also true

Each process keeps an internal counter, that is increment on itnernals ops. On any message it forwards current counter. On message receive it adjust local counter to maximum of internal and received.

This ensures that if event a happened before event b, then a will also be positioned in that ordering before b. Conversely however, $C(a) < C(b)$ does not imply that event a happened before event b.

To capture this *casuality* we need to use **vector clocks**. Let each process p_i maintain a vector VC_i such that

- $VC_i[i]$ is the number of events that occurred so far at p_i, i.e. it is the lampart local logical clock at process p_i
- if $VC_i[j] = k$ then p_i knows that k events have occured at p_j.

We maintain this by:

1. Before executing an event increment internal logical clock $VC_i[i] += 1$
2. When process p_i sends message m to process p_j, its sets $m's$ vector timestamp $ts(m)$ equal to VC_i after having execute previous step
3. When process p_j receives m, it adjusts its own vector by setting $VC_j[k] = max(VC_j[k], ts(m)[k]$ for each k (i.e. merging the casual histories)

Mutual Exclusion

Distributed mutual exclusion algorithms can be classified into two different categories.

- **token based solution** we pass a token message between process, and whoever has it can access shared resource.
- **permission based approaches**, process wanting to access a resource requests permission and waits.

Centralized Algorithm

Use centralized node that coordinates request message for shared resources using a FIFO queue, and grants one at a time.

Distributed Algorithm

Send message for resource type and vector clock. On receive, send back OK if not accessing, queue request without reply if already have access, and finally if receiver is also waiting, compare timestamps of incoming message with internal logical clock and take first one.

The problem: N point of failure, if node fails, it appears like denail of permission.

Token-Ring Algorithm

Simple token based algorithm where processes are put in a ring and circulate token.

Decentralized Algorithm

Each resource is replicated N times. Whenever a process wants to access resource, it will need majority vote from coordinators.

Election Algorithms

Goal: after election, all processes agree on who is the new coordinator

Bully

When any process notices coordinator is not healthy, it initiates election:

- send election message to all process with higher identifiers
- if no one responds, we win the election and become coordinator
- if one of the higher ups answer, it takes over and we are done

Ring

When a process notices coodinator is not healthy it:

- send election message containing its own PID to successor in ring
- if successor is down, sender skips over and moves along the ring
- At each step the sender adds its own identifier to the list in the message making itself a candiadte to be elected as coordinator
- when the message gets back to original, we read the identifier and broadcast who the coordinator was

Consistency

We want to replicate data to increase system reliability, and increase performance.

Definition 20. A **consistency model** is a contract between processes and the data store.

Sequential consistency: the result of any execution is the same as if the read and write operations by all processes were executed in some sequential order appearing the same to each process

Linearizability states that each operation should appear to take effect instantaneously at some moment between its start and completion.

Casual consistency is a weakening of sequential consistency that requires sequential consistency only on potentially casually related writes

Eventual consistency: if no updates take place for a long time, all replicas will eventually become consistent

Montonic read consistency is a *client-centric* consistency model where if a process reads the value of a data item x, any successive read operation on x by that process will always return that some value or a more recent value

Montonic write consistency: a write operation by a process on data item x is complete before any successive write operation on x by the same process

Read your own write consistency: the effect of a write operation by a process will always be seen by a successive read operation on the data item by the same process

Write follows reads consistency: any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process

Replication

A *permanent replica* is a small set of replicas from the beginning. A *server-initiated replica* is typically meant to dynamically load balance, whereas a *client-initiated replica* is typically meant to cache results.

There are three options for state distribution:

1. Propagate only a notification of an update
2. Transfer data form one copy to another
3. Propagate update operations to the other copies

In a **push based protocal** updates are propagated to other replicas without those replicas even asking. Generally applied when strong consistency is required.

- Good when *read-to-update* ratio is high.
- Overhead since Server needs to keep track of clients

In a **pull based protocal** a server or client requests another server to send it any updates it has at the moment. Generally applied for client caches.

- Good when *read-to-update* ratio is low.

Fault Tolerance

Definition 21. Availability is the ability for the system to be used directly

Reliability is the ability for the system to run continuously without failure

Maintability refers to how easily a failed system can be repaired.

Traditional metrics include *mean time to failure*, *mean time to repair*, *mean time between failures*, *availability how many 9's*

Types of failures:

- *Crash Failure*: server prematurely halts
- *Omission failure*: server fails to respond to a request
- *Timing failure*: server response lies outside a specified time interval
- **Byzantine failures**: failures that are incorrect, but cannot be detected as being incorrect

In *resilience by process groups*, we organize several identical processes such that when a message is sent to the group, all members of the group receive it.

Paxos Consensus Algorithm

The assumption under which Paxos operates:

- The distributed system is partially synchronous
- Communication between process may be unreliable
- Messages that are corrupted can be detected as such
- All operations are deterministic
- Processes may exhibit crash failures, but not Byzantine failures

The three stages are:

- Prepare phase: establish the latest generation clock and gather any already accepted values
- Accept phase: propose a value for this generation for replicas to accept
- Commit phase: let all the replicas know that a value has been chosen

Consensus with Byanztine Failures

If we assume replicas only have crash failures, then we need $2k + 1$ servers to reach conensus with k crashes.

In the case with Byzantine failure, reaching consensus will require $3k + 1$ nodes in the presence of k arbitrary failures.

To reach **byzantine agreement** we need:

- every nonfaulty backup process stores the same value
- if the primary is nonfaulty, every backup process stores exactly what the primary had sent

Theorem 0.22. ***CAP Theorem****: Any networked system providing shared data can provide only two of the following three properties:*

- *C: consistency, a shared and replicated data item appears as a single, up to date copy*
- *A: availability, updates will always be eventually exeucted*
- *P: partition tolerance of a process group (due to failure network)*

Patterns

Clock Bound Wait

Wait to cover the uncertainty in time between cluster nodes before reading and writing so values can be correctly ordered across nodes

Problem: if the read request originates in cluster nodes with lagging clocks we lose external consistency. Lamport clocks can only give *partial ordering*.

Solution: cluster nodes wait until the clock values on every node in the cluster are guaranteed to be above the timestamp assigned to the value while reading or writing.

- Google has time service implemented in *True Time*
- AWS has *AWS Time Sync Service*
- Both key clock drift $< 1ms$ in most cases

Consistent Core

Maintain a smaller cluster providing stronger consistency to allow larger data cluster to coordinate server activities without implementing quorum based activities.

Problem: There are many common tasks such as leader election, group membership and routing that needs to be implemented with strong consistency guarantees (linearizability) and fault tolerance. Quorum based consensus is nice but doesn't scale with the size of the cluster

Solution: Implement a small 3-5 node cluster that provides linearizability and fault tolerance. This cluster can manage metadata from the larger data cluster.

Emergent Leader

Order cluster nodes based on their age within the cluster to allow nodes to select a leader without an explicit election

Problem: you don't want to depend on a whole sperate consistent core for leader election.

Solution: Oldest node in the cluster plays the role of the leader or coodinator.

Fixed Paritions

Keep the number of parititions fixed to keep the mapping of data to the partition unchanged when size of cluster changes

Problem: to split data across cluster, there needs to be a mapping that distributed the items uniformly, and such that we can determine which node an item is stored in without checking all nodes.

Solution: map data to logical paritions, map logical partitionts to nodes. Even if nodes add or removed, the mapping of data to logical paritions is left unchanged.

Follower Reads

Serve read requests from followers to improve latency and throughput

Problem: leader may get overloaded, and is subject to additional latency from edge nodes. **Write requests need to go to leader for consistency but reads can be done at the follower node**

Generation Clock

Monotonically increasing number indicating the generation of the server

Problem: The old leader should be able to detect that the rest of the network has assumed a new leader in case the we were temporarily disconnected due to for example our gc.

Solution: Every time a new leader election happens, it should be marked by increasing server generation. Store it in write ahead log across disc failures,

Gossip Dissemination

Use random node selection to pass information through the network and ensures it eaches all nodes without flooding the network.

Problem: A lot of network bandwith and scale issues arise if all nodes need to talk with all nodes. But somehow information needs to propogate everywhere.

Solution: Cluster nodes use gossip style communication to propogate updates, picking random node to pass information to.

Heartbeat

Show a server is available by periodically sending a heartbeat message to all other severs

Problem: timely detection of failures

High water mark

An index in the write ahead log showing last succesfull replication

Problem: write ahead logs can help recover in case server crashes, but are not available. After replicating logs on multiple servers, and using leader and follower pattern, there are still error modes.

Solution: THe HWM is an index into the write ahead log that shows the last log entry that has been succesfully replicated to a quorum of followers.

Hybrid Clock

Combine system clock with logical clock to get version as date-time which can be ordered

Problem: when lamport clock is used, we only have relative order but not actual timestamp of occurence

Solution: combine both into one object

Idempotent Receiver

Identify client requests uniquely so they can ignore duplicate requests when client retries

Problem: Clients re-send request when they don't get response from server, since its impossible to know if the server is down. But if the server crashed after processing the request, we will process duplicate requests which ay not be idempotent

Solution: assign registration unique id to each client, and have each client send requests with unique request id. Server checks if it has processed a request id for a given client id for all non-idempotent instructions.

KeyRange parition

Parition data in sorted key ranges to efficiently handle range queryes

Problem: when access pattern is done by range, naively mapping data items to cluster nodes will be suboptimal

Solution: Create logical paritions of key ranges in sorted order. Then map these partitions to cluster nodes. THen when client asks for range, we can just hit the nodes we need by following the sorted mapping.

Lamport Clocks

Use logical timestamps as a version for a value to allow ordering of values across servers

Problem: Since wall clocks are not monotonic, they cannot be used to determine consistent ordering across servers.

Solution: every server maintains a lamport clock instance which holds an integer value time, which gets incremented via a tick method prior to any write operation, and returns the timestamp that was used for writing the value to the client.

Leader and Follower

Lease

Low water mark

An index into the write ahead log showing which portion of the log can be discarded

Problem: write ahead log maintains every update to persistant storage. It can grow indefinitely over time

Soltion: mechanism gives lowest offset before which the log can be discarded.

Paxos

Quorum

Avoid two groups of servers making independent decisions by requiring majority for taking every decision

Problem: in distributed system we need to replicate result to every server to ensure safety in case of crash. If we wait for full consensus this will not scale.

Solution: A cluster agrees that it has received an update when majority of nodes have ack the update. We call this number the quorum. The size of the cluster minus the quorum indicates how many failure can be tolerated.

Replicated Log

Request Pipeline

Improve latency by sending multiple requests on the connection without waiting for the response of preivous request

Problem: using single socket communication we should improve throughput and latency by allowing multiple requests through to utilize server to max

Solution: asynchronous mechanisms by which one threads sends requests, and another receives requests. This is done asynchronous, and requests are stored locally in waiting queue until full. If full, we resort back to blocking.

Request Waiting LIst

Singular Update Queue

State Watch

Two Phase Commit

Version Vector

Versioned Value

Write Ahead Log

Provide durability guarentee without full data structure being persisted to disc, by instead persistning each state change as a command to an appen only log

Problem: strong durability guarentee are important, but naively are slow

Solution: store each state change as a command on disc in append only form. Each command is given a sequence number id, which can help segment logs, and use low water marks to flush. After crash, we can replay log to get back the full table.

Introduction