Fundamentals

Characteristics:

- Concurrent execution of components
- Independent failure modes
- Transmission delay
- No global time

Implications:

- Can't know why there is no reply
- Can't use local timestamps for ordering distributed events
- Inconsistent views of distributed state
- Can't wait for quiescence to resolve inconsistencies

Design considerations:

- What are the named entities?
- Is communication asynchronous/synchronous, one-to-many/one-to-one?
- Close or loose coupling? Must entities share a programming context or be running simultaneously?
- Replication for load balancing, failure tolerance

Federated administration domains:

- Unit of name and authentication administration
- Inter and intra-domain communication
- Typically per-domain firewall protection
- Policies specified per domain (e.g. for access control)

Dynamic domains:

- Composed of members grounded in static administration domains
- Policies often take the form of contractual obligations
- Trust between members based on observation of audit

Independent external services:

- Naming and authentication may be client-domain related
- Access control related to client roles in domains
- Need for accounting, charging and audit as a basis for mutual trust

Ad-hoc anonymous groups:

- Parties need to decide whether to interact based on trust policy and trust engine
- Policies based on accumulated trust information and cost-benefit analysis

Process groups can have internal structure or none (peer-to-peer), and be closed or open.

Time

UTC is TIA corrected for leap seconds to keep synchronized with the sun.

In a distributed system we must define interval timestamps due to lack of known causality. If intervals overlap they are incomparable.

Clock skew: phase offset between two clocks. Clock drift: frequency offset between two clocks. Accurate clocks decrease the width of interval timestamps.

Lamport time is based on the fact that IPC establishes a partial ordering on events. When you receive an event from another machine, if the time it believes it sent at is greater than your local time, increase you time to the time received plus an increment.

Cristian's algorithm for synchronization with a UTC server:

1. Each machine polls the time server periodically for the time
2. When the time is received, it is adjusted by an estimate for network delay
3. If the time exceeds local time, use it to set the clock or increase interrupt rate
4. If time lower than local time then decrease interrupt rate until target hit

NTP is a hierarchical deployment of Christians algorithm, with UTC at the leaves.

Message ordering can be unordered, total or causal.

Causal order: $send_{p_1}(m) < send_{p_2}(m') \Rightarrow deliver_{p_3}(m) < deliver_{p_3}(m')$

Causal order can be implemented using vector clocks:

- All messages must be multicast to the process group

- Each process records a vector with the most up to date local timestamp of all members of the process group
- Before and after receiving, increment the local process timestamp
- When sending a message attach the current vector clock state
- When receiving a message update the local vector by taking the element-wise maximum with that in the received message
- To implement causal delivery order, if a message arrives with the timestamp for a third party process being strictly greater than the current best known one, wait for an appropriate message to arrive from that third party to fill the gap. This works because we assume FIFO delivery of messages between a pair of processes

Total order can be implemented with a timestamp:

- All messages must be multicast to the process group, including itself
- Receipt of messages is acknowledged as a multicast message to the group
- When acknowledgements for a message have been received from every group member, deliver it locally in timestamp order
- If there is timestamp contention, a tie-breaker is used to choose a message to deliver first

Consistency

Weak consistency is used if fast access is required. A local replica is updated and updates are sent to other replicas. Different replicas may return different item values.

- The order of conflicting updates at different replicas matters. Arbitration needs a policy for resolving conflicts
- Replicas may fail but their updates must not be lost

Strong consistency is used if reliability is required. Protocols ensure that only consistent state can be seen.

- Can be implemented as a transaction: lock everything, make update, unlock everything. Suffers from lack of availability
- Quorum assembly requires locks on $QW > \frac{n}{2}$ machines and $QR > n - QW$ machines out of a total of n. This ensures every quorum contains at least one up to date replica. Upon acquiring a quorum, it is made consistent then the operation performed

For transactions and quorums we need atomic commitment. Two phase commit proceeds as follows:

1. Commit manager (CM) requests and assembles votes from participating sites (PS) and CM as to whether to commit. Sites all secure data and vote
2. CM decides on a commit or abort, which is recorded in a persistent store and propagated
3. If a PS crashes, it must find out state of 2PC from CM upon restart
4. If the CM crashes after recording decision it must tell PSes the decision
5. If any PS does not reply to a vote request, must abort

Need to detect or prevent deadlocks. Detection backs off on quorum acquisition after a timeout, prevention tiebreaks simultaneous quorum assembly requests based on objective metric.

In a large scale system, may have a hierarchy of replicas. Updates propagated down sub-trees after per-level quorum decisions. Correct reads are from top level servers, fast reads are from any other: risk missing recent updates.

Election

With a peer structure we may need to elect a coordinator for external entities to direct requests to.

BULLY:

1. P notices no reply from the coordinator
2. P sends ELECT to all processes with higher IDs
3. If any reply, P stops trying
4. If non reply, P becomes the new coordinator and sends COORD to the group
5. If receive ELECT at any time, reply with OK and initiate election if necessary

RING, applicable if processes are ordered into a well-known ring:

1. P notices the coordinator is not functioning
2. P sends ELECT tagged with its own ID around the ring
3. If receive ELECT at any time, append own ID if not present and pass on, else send COORD electing the highest ID in the ELECT message as the coordinator

Both of these methods elect the surviving process with the highest ID. This ensures that even if multiple elections run concurrently they all agree to the same coordinator.

Distributed Mutex

Centralized algorithm: a process is elected as coordinator and grants lock requests. This is fair and economical, but creates a bottleneck and a single point of failure.

Taken ring: a token giving permission to lock circulates indefinitely. This is not fair and we must handle token loss, but is quite efficient.

Distributed: send timestamped request to all processes, when they all reply say you have the lock. When receiving such a message, defer reply until you are locally unlocked unless you are executing a request, in which case use timestamps to work out whether you should reply granting permission or wait for your own grant to be given and completed instead. This is fair, but has multiple points of failure and bottleneck, uses many resources and it is ambiguous what a lack of reply means.

Middleware

Middleware is a layer between the OS and applications that hides complexity and heterogeneity of distributed systems. Typically supports naming, location, service discovery, replication, communication faults, QoS, synchronization, concurrency, transactions, access control and authentication!

- Request/reply vs. asynchronous messages
- Language specific vs. agnostic
- Tightly vs. loosely coupled

RPC masks remote function calls as local:

- Provides marshalling and unmarshalling of function arguments and return value
- Synchronous request/reply paradigm (built in synchronization)
- Easy to understand for programmer
- Distribution transparency if there are no failures
- Can get delivery "at most once" (programmer may retry) or "exactly once" (hard error return upon failure)
- Tight coupling between client and server
- Lacks notion of services

OOM allows objects and references to them to be local or remote:

- Very similar to RPC
- Supports OOP model
- Can provide location transparency via object references
- Has notion of service (interface), persistent objects
- May have to perform distributed garbage collection

Message Oriented Middleware (MOM) communications through messages stored in queues:

- Asynchronous interaction and message server component can decouple client and server
- Supports reliable delivery service
- Can process/filter/transform messages in the middleware
- Programming abstractions typically poor
- Request/reply interactions difficult to achieve
- One-to-one communication can limit scalability

Web Services:

- Use well known standards for distributed computing (HTTP, SOAP, XML, WSDL)
- UDDI allows service discovery
- Synchronous and asynchronous messaging

Event-Based Middleware has publishers and subscribers in many-to-many communication:

- Topic and content based (infrastructure filtered) publish/subscribe
- Asynchronous interaction and middleware decouples client and server
- Many-to-many interaction very scalable
- Topic and content based filtering very expressive
- Some applications may need to know the sender or receivers identity
- Request/reply interactions difficult to achieve and impossible in general due to sender anonymity

Composite event detectors increases expressiveness of content-based publish/subscribe by allowing patterns of events to be found and published as composite events.

Naming

Pure names yield no information about the thing named.

Unique identifiers (UIDs) are never reused (are immutable references). This can be achieved with a hierarchy or bit patterns. However, hierarchy may not be appropriate for pure names.

Name space: the collection of valid names recognized by a name service.

Naming domain: a name space for which there exists a single overall administrative authority for assigning names within in.

Name resolution: obtaining a value which allows an object to be used.

Name servers hold a mapping from object type and name to a list of attribute values.

To do hierarchical resolution, the user agent starts with the name server root address and iteratively resolves parts of the hierarchical namespace. They may also ask servers to do the resolution recursively to build up server-side caches.

DNS:

- Computers using DNS are grouped into zones, within which management of sub-domains is delegated
- A zone has a primary name server (authority), and multiple secondary servers holding replicas

If system wide consistency is guaranteed we have to delay on update and lookup. However, due to fast access requirement we just do weak consistency. This is justified because in some cases we have a means of detecting obsolete naming data (e.g. address fails to work), naming data doesn't change very fast and changes propagate quickly, and if it works it doesn't matter that it's out of date!

Note that even in a weakly consistent system it is desirable that we have long term consistency: if updates stopped we would eventually have consistency. This can be achieved by transmitting whole directories periodically and comparing them.

Access Control

Conceptually, an access matrix of principals against objects holds rights in its elements.

ACLs are a principal name and access rights:

- Allow subtle expression of policy (on principal or group basis, with exceptions)
- Slow to check, so don't scale: need to scan principal list
- Awkward to delegate rights
- Easy to revoke, but may have delayed effect if only check upon resource open

ACLs can be checked at the home OS and this trusted by the server.

Capabilities are an object name and access rights:

- Quick to check, so scale well
- Must prevent authorized creation, tampering and theft
- Can implement delegation easily, either controlled or free
- Possible to do selective and quick revocation

Capabilities can be constructed by taking check digits to be $f(\text{SECRET}, \text{protected fields})$. The SECRET is known only to the object manager, which is queried by services checking a capability. This protects against tampering, but not propagation, and revocation will either be slow (check hot list) or indiscriminate (change SECRET).

By adding an unforgeable principal name to the protected fields, we can additionally protect against theft and propagation, but means that delegation must be explicit.

By using the principal name in the encryption function only we can obtain the above properties as well as anonymity at the invoked service.

Fast delegation can be done by using timeouts on issued capabilities and transient hot lists of revoked capabilities that have yet to time out.

RBAC generalizes groups in ACLs to roles:

- Clients of services are classified into role types, which may be shared between multiple services
- Access rights are assigned to roles for use of the service
- Separates the of administration of principals into groups/roles and service's specification of authorization policy
- Roles ease administration of inter-domain authorization policy as roles change less often than principals and administration of principals is separate from service use
- Some standards use hierarchical roles with privilege inheritance, but this is difficult to use to model an organization like this and you cannot separate duties or use least privilege
- Can be extended to include roles that capture exclusions and other relationships
- Can be implemented using certificate technology to prove you inhabit a role

Storage Services

Possible requirements:

- Composite documents with components of different media types
- Linking related information across files
- Applications for download into home/other systems
- Access to files from remote locations
- Detached operation
- Use of spare capacity with peer-to-peer

Characteristics:

- Open/closed (is it bound into single OS file system?)
- Level of interface: block level/UID named files/path-named files
- Whether the service is responsible for managing caches and replications of files
- Stateless or not? Aids crash recovery but impedes ability to manage caches and do concurrency control
- Existence control: a file should stay in existence for as long as it is reachable from the root of the directory naming graph
  - A directory service can do existence control for its own objects provided they cannot be shared
  - Lost object problem: memory of object erased somewhere by server or client crash
  - Could have "touch" operation that must be periodically used by clients to prevent deletion

CFS:

- Indexes used by clients to mirror their directory structures, entries point to other indexes and files: atomicity guaranteed
- Existence control done using reference count from indexes, with asynchronous garbage collector
- MRSW concurrency control, unlocked upon timeout
- Can extend storage type system with structure sufficient to locate embedded links: this allows existence control to take account of links between files