Distributed Systems

Some common phenomena in distributed systems

1. Unpredictable delays (communication)
   - Arise due to:

2. Missing or improper time-base
   - (usually due to temporal variation)

3. Partial failures
   - (if all failures are individual failures, then it is an event of a good design)

Time in distributed systems

Two alternative strategies:

Based on a shared time
Synchronize clocks!

Based on sequence of events
Create a virtual time!

Virtual (logical) time

Notion of concurrency:

\[ a \prec b \Rightarrow \exists x \in \mathbb{R} \text{ such that } a < x < b \]

Notion of causality:

\[ a \prec b ; \text{Send} \ (\text{message}, (a, b)) \]

Virtual (logical) time

\[ a \prec b \Rightarrow \exists x \in \mathbb{R} \text{ such that } a < x < b \]

Causal order:

\[ (a, b) \prec (c, d) \Rightarrow a < c \lor \ (a = c \land b < d) \]

Virtual (logical) time

\[ a \prec b \Rightarrow \exists x \in \mathbb{R} \text{ such that } a < x < b \]

Causal order:

\[ (a, b) \prec (c, d) \Rightarrow a < c \lor \ (a = c \land b < d) \]

Virtual (logical) time

\[ a \prec b \Rightarrow \exists x \in \mathbb{R} \text{ such that } a < x < b \]

Causal order:

\[ (a, b) \prec (c, d) \Rightarrow a < c \lor \ (a = c \land b < d) \]

Virtual (logical) time

\[ a \prec b \Rightarrow \exists x \in \mathbb{R} \text{ such that } a < x < b \]

Causal order:

\[ (a, b) \prec (c, d) \Rightarrow a < c \lor \ (a = c \land b < d) \]

Virtual (logical) time

\[ a \prec b \Rightarrow \exists x \in \mathbb{R} \text{ such that } a < x < b \]

Causal order:

\[ (a, b) \prec (c, d) \Rightarrow a < c \lor \ (a = c \land b < d) \]
Distributed Systems

Distributed critical regions with logical clocks

- \( \forall \text{times} \); \( \forall \text{requests} \);
  - \( \text{Add to local RequestQueue ordered by time} \)
  - \( \text{Reply with Acknowledge or OnRequest} \)
- \( \forall \text{times} \); \( \forall \text{Release messages} \);
  - \( \text{Delete corresponding request in local RequestQueue} \)
  - \( \text{Add OnRequest to local RequestQueue (ordered by time)} \)
  - \( \text{Send OnRequest to all processes} \)
- \( \text{Wait for all} \text{OnRequest} \text{and no outstanding replies} \)
- \( \text{End} \text{text input} \text{critical region} \)

Analysis
- No deadlock, no individual starvation, no livelock.
- Minimal request delay: \( N \) - 1 requests (1 broadcast) + \( N \) - 1 replies.
- Minimal release delay: \( N \) - 1 release messages (1 broadcast).
- Communications requirements per request: \( 3N \) - 1 messages (or \( N \) - 1 messages + 2 broadcasts).
- Clocks are kept synchronized by exchanging messages themselves.

Assumptions:
- \( \text{Messages are fast} \)
- \( \text{Solutions must be} \)

Distributed Systems

Distributed critical regions with a token ring structure

1. \text{Organize all processes in a logical or physical ring topology}.
2. \text{Send an initial message to one process}.
3. \text{N times, \( \text{N} \) processes can receive the token message}.
4. \text{If no one responds after a predefined amount of time:} 
   - \( \text{Pi waits for the next message} \)
   - \( \text{A global, central coordinator is employed in some systems... yet} \)
   - \( \text{if it fails, a system to come up with a new coordinator is provided} \)

Distributed Systems

Electing a central coordinator (the Bully algorithm)

Any process \( \Pi \) which knows that the central coordinator is gone, proceeds:

1. \text{Find an Election-message to all processes with higher process numbers}.
2. \text{Produce and exchange an Election-message to \( \Pi \)}.
3. \text{If \( \Pi \) receives an Election-message from \( \Pi' \), it deletes its own and sends a Coordinator-message to \( \Pi' \)}.
4. \text{Any process \( \Pi' \) that receives a Coordinator-message waits for an Election-message from a process with a higher process number, or requests from the originating process.}

Snapshot algorithm

- \text{Once in processor, process creates a snapshot of its local state, \( s_\Pi \)
  - \( \mathbf{f} \text{,} \) \( \Pi \text{'s} \) state at the moment of the snapshot.
  - \( \mathbf{g} \text{,} \) \( \Pi \text{'s} \) history up to the moment of the snapshot.
  - \( \mathbf{h} \text{,} \) \( \Pi \text{'s} \) messages that are to be sent to other processes.
  - \( \mathbf{k} \text{,} \) \( \Pi \text{'s} \) messages that are to be received by other processes.
  - \( \mathbf{l} \text{,} \) \( \Pi \text{'s} \) messages that have been sent but not yet received.
  - \( \mathbf{m} \text{,} \) \( \Pi \text{'s} \) messages that have been received but not yet processed.
  - \( \mathbf{n} \text{,} \) \( \Pi \text{'s} \) messages that have been received and processed.
  - \( \mathbf{o} \text{,} \) \( \Pi \text{'s} \) messages that are to be forwarded.

Running the snapshot algorithm: 

- \text{Each processor \( \Pi \) produces a set of messages, \( s_\Pi \), \( \Pi \) sends to local site, \( s_\Pi \).
  - \( \Pi \text{'s} \) state at the moment of the snapshot.
  - \( \Pi \text{'s} \) history up to the moment of the snapshot.
  - \( \Pi \text{'s} \) messages that are to be sent to other processes.
  - \( \Pi \text{'s} \) messages that are to be received by other processes.
  - \( \Pi \text{'s} \) messages that have been sent but not yet received.
  - \( \Pi \text{'s} \) messages that have been received but not yet processed.
  - \( \Pi \text{'s} \) messages that have been received and processed.
  - \( \Pi \text{'s} \) messages that are to be forwarded.
Distributed Systems

Snapshots algorithm

Termination condition:

- If: Make assumptions about the communication delays in the system.
- <consistent events> are sent and received messages, for all processes. Include this in the state, ordering both received messages in the observer process.

Distributed states

- Sorting the events into past and future events.
- Past and future events uniquely separated or consistent state.

Distributed systems

- Consistent distributed states

  Why would we need that?
  - Find deadlocks
  - Find termination / completion conditions
  - Any other global safety of liveness property
  - Collect a consistent system state for system backup/restore.
  - Collect a consistent system state for further processing (e.g., distributed database).

Transactions

- Concurrency and distribution in systems with multiple, independent interactions?
- Concurrency and distributed client/server interactions beyond single remote procedure calls?

Transactions (ACID properties)

- Atomicity: All or none of distributed suboperations are performed.
  Atomicity helps detect a crash condition for a client or server, thus it is possible for a rollback procedure to be executed before the transaction is committed.
- Consistency: Ensures the same state is maintained after an update to a distributed system.
  Consistency is not revealed unless the transaction commits.
- Isolation: Isolates transactions from other concurrent transactions.
  Isolates results including partial results not revealed until the transaction commits.
- Durability: Transactions ensure that once a transaction commits, the transaction commits if the operation on the same object is not interrupted by another operation on the same object.
  Durability means data gets written to a state before the transaction is completed.
Three sequential phases:

1. Read a work
   - Create a shadow copy of all involved objects and portions of required operations on the shadow copy and locally it is serialization.
   - Validate whether operations are correct and check if no errors occurred due to concurrency.
   - Obtain write grant.

2. Keep the local
   - Write to the local database.
   - After local commit check all occurred interleavings and check for serializability.

3. Update or abort
   - If serializability could be ensured in step 2 then all results of the involved transactions are written locally.
   - Otherwise, destroy the shadow copies and abort any work in the involved transactions.
Handling faults in distributed systems typically involves redundancy (replicated servers), distributed transaction schedulers, and distributed processing.

1. **Redundancy (replicated servers)**
   - **Start-up (initialization) phase**
     - Coordinator initializes the system.
   - **Processing starts**
     - Coordinator determines the next job to process.
     - All servers process the job.
   - **Result delivery**
     - Coordinator reports the result.

2. **Distributed transaction schedulers**
   - **Phase 2: Report result of distributed transaction**
     - Premise: A reliable, high-availability computer system should be implemented for key parts of the system.
     - Example: File server.
     - **Optimistic methods**
       - Reduces overheads; avoids deadlock prevention.
     - **Locking methods**
       - Ideal for single-server systems.

3. **Redundancy (replicated servers)**
   - **Stages of each server**
     - Coordinator sends a job message to all servers.
     - All servers process the job.
     - Coordinator determines the next job to process.
     - Coordinator sends the job message to all servers.

4. **Summary**
   - Distributed systems:
     - Networks: OSI, topologies, practical network standards.
     - Time: Synchronization, distributed processing, distributed computing.
     - Distributed systems: Distributed transaction schedulers, distributed processing, distributed computing.