Disposition 7: Consistency

Mathias Ravn Tversted

January 14, 2020

Table of contents

Unstructured Peer to Peer

Flooding Network

A flooding network is a network where messages are flooded. This guarantees that if a message is sent, it will eventually be delivered to all correct processes. When receiving a new message, it is simply sent to all other peers. This means that we cannot guarantee total ordering since messages can overtake each other.

Flood Protocol (Ideal Functionality)

Send: Process P_i gives input of form (P_i, m) on user port $Flood_i$ **Deliver** Party P_i gets output of form (P_j, m) on user port $Flood_j$. Message m sent by P_J was delivered to P_i by (flood) We assume that the system has the following properties **User Contract**: We require from the process P_i that it does not send the same message twice. When the user keeps its contract, we require that the flooding system has the following safety and liveness properties. We also require that in every finite time interval a party sends at most a finite number of messages. **Safety**: If a correct P_j outputs (P_i, m) then earlier P_i sent (P_i, m) **Liveness**: If correct P_i sends (P_i, m) then eventually all correct P_j deliver (P_i, m)

Consistency models

FIFO

First In, First Out (FIFO)

We want to preserve the order of communications. We can do this through FIFO (First In, First Out). **FIFO**: If a correct P_i sends (P_i, m) and later sends (P_i, m') then it holds for all correct P_j that if they deliver (P_i, m'), then they delivered (P_i, m) earlier. User Contract and Liveness is the same as for *Flood*.

FLOOD2FIFO: This Time It's Protocol

- Each party P_i sets counter $c_i = 0$. This counts number of *messages sent*ialises n counters $r_{i,j} = 0$ for $j = 1, ..., n$ These track how many messages P_i received from P_j
- P_i: When sending message m, let $c_i = c_i + 1$. Send P_i, c_i, m on floodin network. So tag each message with seq number.
- P_i. When receiving (P_j, c, m) store it in buffer until $r_{i,j} = c + 1$. Now let $r_{i,j} = r_{i,j} + 1$ and output (P_j, m). Since $r_{i,j}$ is the number of messages P_i received from P_j, if $r_{i,j} = c + 1$ then (P_j, c, m) is the next message.

Casuality

Causality

Just because messages arrive in FIFO order, does not mean that this is a meaningful order. *Casual order* looks at which messages could have *caused* other messages. This could be things such as messages in a messaging application. But we need *vector clocks*, before we can proceed.

Vector Clocks

For a system of n parties, a *vector clock* is a vector $VectorClock \in \mathbb{N}^n$ where $VectorClock[P_j] \in \mathbb{N}$ where that is the entry associated with party P_j. Since each party has a vector clock, VectorClock(P_i) is the vector clock of P_i. $VectorClock(P_i)[P_j] = s$ means that P_i knows that P_j has sent s messages.

- When P_i sends message m it increments $VectorClock(P_i)[P_i]$ by one
- When P_i sends message m it sends along $VectorClock(P_i)$. When P_i sends m that message can be influenced by exactly the messages that have influenced P_i at the time P_i sends a message.

Vector Clocks

- By sending the vector clock with the message, you can determine which messages could have influenced m
- Formally: $s = VectorClock(P_i, m)[P_j]$ messages from P_j could have influenced m

Vector Clocks

- Any receiver P_r knows that it is safe to deliver (P_i, m) once it delivered s messages from P_j it is s that could have influenced m
- To keep track of how many messages were delivered from each party P_j keeps a vector clock $Delivered(P_j)$, where $Delivered(P_j)[P_i]$ is how manymessages P_j delivered from P_i. This gives the rule that P_j can deliver (P_i, m) once it holds for all P_k that $Delivered(P_j)[P_k] \geq VectorClock(P_i, m)[P_k]$

Causal-Past Relation

There is a notion that certain messages could have *caused* other messages. Or "the happens-before relation".

Kasusel-past relation: $(P_i, m_i) \hookrightarrow (P_j, m_j)$

If there is a chance that m_j depends on m_i, then we treat it as if that is the case.

Denote $CasualPast(P_j, m) = \{(P_i, m_i) | (P_i, m_i) \hookrightarrow (P_j, m_j)\}$ the set of things that may depend on (P_i, m_i).

We can similarly define a *Kausual Future*.

C-ass-ual network: In Practice

- **Ports**: It connects n parties. For each party there is a port $Casuel_i$. It has ports leak and deliver
- **Init**: $\forall P_i$ keep $Delivered_i = Sent = \emptyset$
- **Send**: On input (P_i, m) on $Kasusel_i$.
  $Sent = Sent \cup (P_i, m)$.
  $KasuselPast(P_i, m) = CasualPast(P_i) \cup (P_i, m)$
- On (P_j, P_i, m) where $(P_j, m) \in Sent$ but not Delivered. If $KasuselPast(P_j, m) \subset Delivered_i \cup (P_j, m)$ then deliver it and update Kasualpast.

Total order

Total Order

Total Order: If correct P_k delivered (P_i, m) and laterdelivered (P_j, m'). Then it holds for all P_m that if they deliver (P_j, m') they delivered (P_i, m) earlier.

We can implement this using Kasusel ordering and Vector Clocks.

TOB from Casuselss and shizzle

- **Init**: For each P_i it keeps $InTransist = UnOrdered\emptyset$
- **Send**: On input (P_i, m) on $Flood_i$ output (P_i, m) and add $(P_i, m) \rightarrow Unordered$
- **Order**: On input (P_i, m) on Order, if $(P_i, m) \in Unordered$ then pop and add it to the queue $InTransit_j$
- **Deliver**: On input P_i on Deliver, pop *InTransit* and output it.