Outline

- Asynchronous model and Motivation for seeking alternatives
- An alternative model for managed environments and a design approach
- An alternative design approach for the Asynchronous model
Asynchronous Delay Model

- Two connected operative processes
- One sends a message $m$ to the other
- How long will it take for $m$ to be received?
  - Communication delay cannot be bounded with certainty
- How long will it take to process the received $m$?
  - Processing delay also cannot be bounded with certainty
- Asynchronous model captures environments, where
  - Processing loads and network traffic can fluctuate by arbitrary amounts at arbitrary instances,
  - Processes’ clocks cannot be kept synchronised (free of time)
Cost of Asynchrony: where and why

- Some critical services are always needed
  - E.g., Chubby Lock Service
- Service replication against host failures
- State updates must be done in an identical order at all operative replicas
- Ordering update requests ≡ strong consistency
- Asynchronous ordering is expensive due to this (FLP) dilemma:
  - A process waits on a timeout and timeout expires
  - Does it mean a failure or timeout duration was too small?
- Cause of ‘performance bottleneck’ in Paxos
Alternative to Asynchronous model

- Emergence of Managed Environments
  - Cluster computing, Data-centres
- Do delays fluctuate so arbitrarily here?
- With Proactive measurements, delay bounds can be predicted in probabilistic terms
- In **probabilistically synchronous** model, the following are known
  - Loss probability,
  - Delay distribution,
  - Jitter
- Claim
  - We can design protocols, minimising the likelihood of having to go the Paxos way for order/strong consistency
The hypothesis behind the new model

- The central hypothesis
  - Most of the time, performance in recent past is indicative of performance to unfold in near future

- Inspiration: congestion control
  - RTO expires $\Rightarrow$ multiplicatively reduce transmission rate
  - RTT and variations in RTT (jitter) are proactively measured and are assumed to hold now
  - Assumes adherence to the same hypothesis
Design Steps

- Measure delays proactively and predict delays in probabilistic terms
- Design protocol with tuneable parameters
- A Schema for run-time choice of parameter values
  - probability of correct ordering is chosen
- Mistakes occurring are detected
- Exceptions on detecting mistakes
Order Protocol – a very brief sketch

- For brevity, assume
  - sites fail by Crash
  - clocks are synchronised
  - messages are not lost (not so in the paper)

- \( P_0, P_2, \ldots, P_n \) are stateful replicas

- Say, \( P_0 \) receives an update request

- It sends \( m \) twice to \( P_1, P_2, \ldots, P_n \):
  - copy 0 at time \( t \) and copy 1 at \( t+\eta \);

- Each of \( P_1, P_2, \ldots, P_n \) also sends \( m \) twice, if it does not receive copy 1 within a timeout;

- Every \( P_i \) (including \( P_0 \)) applies update in \( m \) at time \( t+D \)
Value of D

- Evaluated for the *desired* probability of correct ordering
  - can be chosen to be arbitrarily close to 1
- $D$ is also a function of
  - Measured delays – fact of life
  - Number of ‘nasty’ crashes expected *while* $m$ being ordered
    - A value of 1 is safe and 2 is optimistic
- In Paxos, $(t+D)$ is when
  - a majority of processes *are known to* have settled on the same order number for $m$
- What if $D$ used happens to be small?
  - All operative $P_i$ ‘eventually’ receive $m$
  - Incorrect ordering is detected for initiating exception
  - In PL experiments, no incorrect ordering when there are no ‘nasty’ crashes [8]
So, the full picture

- With a chosen probability $p$, run the order/consistency protocol
  - Wait for $D$ and act
- Inconsistencies occur with $(1-p)$
- Detection assured
- Deal with inconsistency in an application specific way
- In the extreme, exception handler will have Paxos-like complexity + potential roll-back
Crash-Signal Abstraction

- What if the hypothesis cannot hold most of the time?
  - Say, due to malicious (or seemingly malicious) activities
- Say, a process were to signal prior to crash
- Timeout-based failure detection not needed
- For crash-signal, we need
  - A pair of order processes checking each other
  - And a trusted link connecting the pair
  - for the same node redundancy as BFT
Conclusions

- In managed hosting environments, delays are
  - Neither synchronous (can be bounded with certainty)
  - Nor asynchronous (cannot be bounded with certainty)
- They are *probabilistically synchronous*
  - Can be bounded with certainty *most of the time*
- On-going work: development of exceptions
- Open environments are asynchronous
  - On-going work: Crash-signal Menicus
Questions..