PUSH TECHNOLOGY™

Principles of Reliable Communication & Shared State

What Can Go Wrong & Why You Should Care

Dr Andy Piper – CTO, Push Technology

andy@pushtechnology.com

QCon

Please evaluate my talk via the mobile app!

About Me?

- Dr Andy Piper
- CTO at Push Technology
- Ex-BEA/Oracle
  - Architect for WebLogic Server Core
  - Architect and then Engineering Director for Oracle Event Processing
- Spring contributor and Author
- Contributed to many standards – OMG, JCP, OSGi Alliance
- PhD, Cambridge, Distributed Systems
- MBA, Warwick Business School

Agenda

- Communication failure modes
- Basic fault tolerance
- Process failure and resilience
- Classic problems
- Reliable group communications

Communication

PUSH TECHNOLOGY

What Can Go Wrong?

?

PUSH TECHNOLOGY

What Can Go Wrong?

- Crash failure or fail-stop
  - A participant disappears, but was ok up until then. Server reboot, JVM crash, data centre flood.
- Omission failure
  - A participant fails to respond to an incoming message
  - Receive omission
    - A participant fails to receive incoming messages. Network failure, buffer overflow.
  - Send omission
    - A participant fails to send messages. Network failure, buffer overflow, queue overflow.
- Timing failure
  - A participant's response is outside the specified time interval. Stop-the-world GC
- Response failure
  - A participant's response is incorrect
  - Value failure
    - The value of the response is wrong. JVM bugs, concurrency bugs, over-zealous proxies.
  - State transition failure
    - A participant deviates from the correct flow of control. Application bugs, concurrency bugs.
- Arbitrary or Byzantine failure
  - A participant produces arbitrary responses at arbitrary time. Malicious intent. Application retries with network retries.

[Cristian, 1991; Hadzilacos & Toueg, 1993]

Why Does it Matter?

"Reports that say that something hasn't happened are always interesting to me, because as we know, there are known knowns; there are things we know that we know. There are known unknowns; that is to say, there are things that we now know we don't know. But there are also unknown unknowns – there are things we do not know we don't know."

—United States Secretary of Defense, Donald Rumsfeld

- The key question is "What effect does failure have on the state of my system?"
- Actions and failures without consequences can be ignored
- Actions and failures with consequences cannot be ignored
- Often difficult to tell which is which and what the consequences are
- **Theoretical models which prove certain outcomes under certain conditions are therefore *essential***

Law of Unintended Consequences

- Robert K. Merton 1936
  - Ignorance – impossible to anticipate everything (complexity)
  - Error – incorrect analysis (complexity)
  - Immediate interest – not considering long-term outcomes
  - Basic values – prohibition of certain outcomes
  - Self-defeating prophecy – fixing a problem that's not there
- Without sound theoretical models the cure can be worse than the problem
  - Addition of complexity to solve unanticipated edge cases increases overall risk of unknowns and hence failure
    - The Rumsfeld effect
  - Theoretical models are provably able to deal with all circumstances within the scope of the defined problem
  - My own experience
    - WebLogic clustering, sliding windows circa 1999 – major outages at globally visible brand
- But … sound theoretical models can be really hard to implement
  - My own experience
    - TOTEM implementation for WebLogic Event Server circa 2006
- Generally KISS is always good where reliability is concerned

Coping With Failures – aka Fault Tolerance

- We can't actually prevent failures – only mask the fact that they occurred
- If a failure can occur it generally will occur at some point and you have to either cope or not care
  - Not caring is a cheap, viable strategy
  - If you care then basing decisions on the happy path is largely pointless, get over it!
- The key to masking faults is redundancy
  - **Information redundancy** – adding extra information to allow the original to be recovered
  - **Time redundancy** – actions can be re-performed until success, e.g. transactions
  - **Physical redundancy** – extra pieces are added in hardware or software to cope with loss

Reliable Communication

- It helps if your transport is reliable
  - TCP/IP – reliable
  - UDP – not reliable
  - UDP Multicast – not reliable
- Reliable transports mask omission failures
- Unreliable transports do not!
  - Not necessarily bad - some algorithms are tolerant of unreliable transports – e.g. TOTEM
- Crash failures are not masked by reliable transports
  - Most algorithms focus on masking crash failures

Acks

Adrian?

Hi!

Acks - What Can Go Wrong?

Acks – What Can Go Wrong?

1. The server cannot be found
2. The request message is lost
3. The server crashes after it receives a request
4. The ack/reply is lost
5. The client crashes after sending a request

- If the server cannot be found then we cannot proceed, but we know nothing bad happened
- If a request message is lost we can resend after a certain time period
  - Requires that the server deal with retransmissions if it wasn't actually lost
- If the server crashes we need to know whether it did something before it crashed

What Did The Server Do?

Adrian?

Adrian?

We can't retry

We can retry

What Did The Server Do?

- Essentially we don't know from a client perspective
- So what should we do?
  - Three schools of thought

- Always retry – **at-least-once** semantics
- Never retry – **at-most-once** semantics
- Make no guarantees

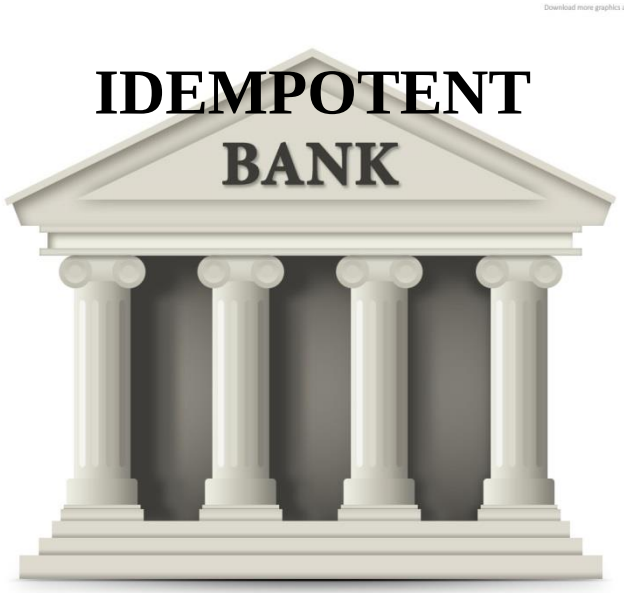
- None of these are attractive
  - We really want **exactly-once** semantics
  - But in general we cannot arrange this

- Lost replies have similar problems
  - Cannot distinguish between a reply that was truly lost and is just slow

Idempotency

- One way round these issues is to guarantee that making a request multiple times is safe
- This is called **idempotency**
- If requests are idempotent they can always be retried and at-least-once semantics save the day

- For many distributed problems this works well, but not all requests can be idempotent



"We will transfer your money at least once"

Sequencing

- Another way of de-duping multiple retries is to add sequence numbers to messages.
- This adds a per-client overhead and, more crucially, **state** to the server that must be maintained across a crash
- Sequence numbers do not resolve lost replies, they must always be sent again

Reliable Communications to Fault Tolerant Systems

- Reliable communications is a building block for reliable systems as a whole
- Often systems builders are interested in **data** rather than communication
  - Communication is a way of manipulating data
- Reliable systems need sender and receiver to be reliable, not just the communication layer
- In many circumstances reliable communication is not even a pre-requisite
  - For a reliable system as a whole

- We define a system as **k fault tolerant** if it can survive k faults before failing

Process Resilience

- Making sender and receiver resilient requires process resilience
- The design principle is to create a group of homogenous processes that can be used to tolerate process failure
- For well behaved processes it is sufficient to have k+1 members to be k fault tolerant
- When a message is sent to a group all members receive it
- If a process fails another can take over because all have the same world view
- This simple premise involves a number of challenges
  - Group membership needs to be maintained
    - Membership can either be **agreed** or **coordinated**
    - However leader election generally must be agreed anyway or become a **single-point-of-failure**
  - Group membership needs to be synchronized with messages
    - Don't send messages to crashed members
    - Do send messages to new members
    - Otherwise your world view is corrupted
  - Group formation must survive catastrophe
  - Managing groups requires communication!

- How easy is it to reach agreement?

Agreement and the Two-Army Problem

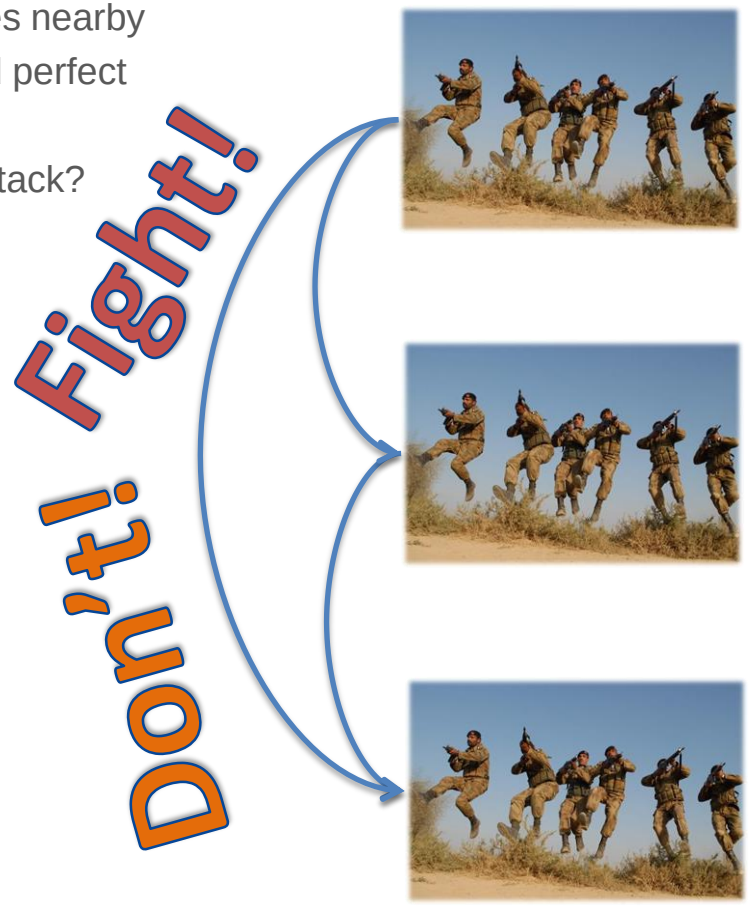
- Famous problem – the **two-army problem**
- The red army has 5000 men
- The blue army has two groups of 3000 men
- If the blue army can coordinate its attack it can win
- If either blue army attacks on its own it will be wiped out
- Communication is inherently **unreliable**

Agreement (or Consensus)

- Neither party can ever be sure that the other is going to do what they expect
  - No matter how many steps the last one is always the decider
- **Agreement between two parties is impossible** even if the parties are perfect
  - Note that only communications is unreliable, the parties always do the right thing

Agreement and the Byzantine Generals

- Another classic problem
- Let's now assume that communications are reliable but the armies are not
- Still one red army but *n* blue generals head armies nearby
- Communications are pairwise, instantaneous and perfect
- *m* of the generals are traitors!
- Can the loyal generals reach agreement on an attack?



Agreement and the Byzantine Generals

- The generals **can** reach agreement, but only if enough of them agree

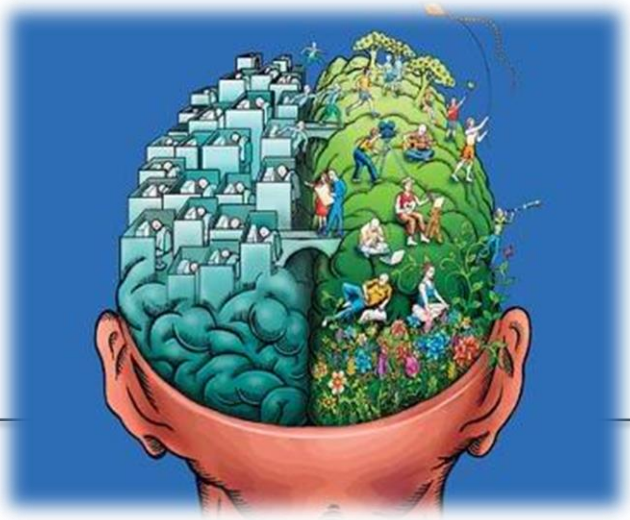
- Lamport (1982) showed that in general agreement can be reached but only if 2m+1 correctly functioning processes are present
  - Thus you need to have a minimum of 3m+1 processes to tolerate m faulty processes
  - In other words **more than 2/3's** need to be working – hence a **minimum of 4**
  - 100% reliability requires infinite processes
- Lamport also provided an algorithm that achieved this in certain circumstances

- Agreement is an active area of research and several famous algorithms exist
  - The key is reaching agreement in as short a time as possible while minimizing communications
  - For example **Paxos** – a Lamport invention, used by Google Chubby/Spanner, Heroku etc
  - Another example is the **Chandra–Toueg** consensus algorithm

- **http://www.andrew.cmu.edu/course/15-749/READINGS/required/resilience/lamport82.pdf**

Consensus and Split Brain

- How common is byzantine failure?
- Actually pretty common when dealing with network partitions
- If a network partition occurs in a process group both sides of the partition can believe they are the only ones alive
  - Leads to inconsistent updates/messages
  - Commonly know as **split-brain** behaviour
- Consensus is the only reasonable way of dealing rigorously with split-brain
  - Typically consensus is separately reached on what side is active and the minority side **fences** itself
- Consensus is an expensive way of dealing with split-brain
  - Potentially a significant number of servers cannot be used
  - It's easy to partition in a way that there are more than m faults
  - Many systems instead **optimistically** ignore split-brain and reconcile updates on re-joining
    - E.g. Hazelcast



Consensus and Failure Detection

"Quis custodiet ipsos custodes?"

- Read the small print!
- Most consensus algorithms assume failure detection
- Failure detection is a non-trivial problem
- Failure detectors can themselves suffer from failure
  - False positives (especially with fail fast) and false negatives
  - “Glitches”, for instance GC, can look like failures
- Some algorithms assumes particular characteristics of failure detectors
- Some algorithms have failure detection built-in

FLP Impossibility

- **Fischer, Lynch and Paterson** proved that consensus in a fully asynchronous system cannot be guaranteed within bounded time
- In practice this is highly unlikely to occur…
- …but illustrates the point that ultimately there are no guarantees

- …But wait - I thought you said Lamport proved it was possible with 3m+1 processes
- Yes – **possible**, but not **guaranteed**
  - in other words algorithms will sometimes not terminate

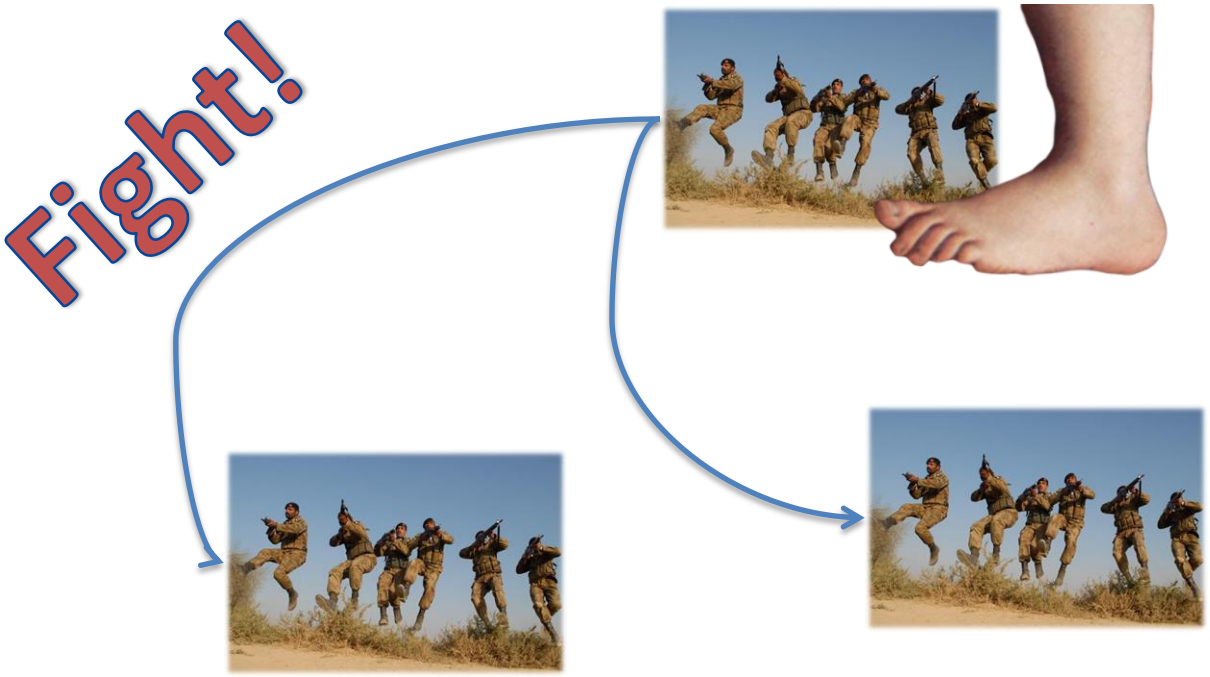
- **http://www.ict.kth.se/courses/ID2203/material/assignments/misunderstanding.pdf**

Group management to reliable group communication

- Group management using consensus (or other means) is a necessary precondition to:
  - Masking process failure
  - Communicating reliably in a group
- Once we know who comprises a group how do we use that information to communicate reliably?

Reliable Group Communications

- In order to ensure that processes maintain the same world view you want to be able to communicate reliably with them as a group
- Messages must either reach all or none of the non-faulty processes and each must know this
  - Thus if a process fails the others know what it received and what to send it if it re-joins
  - Absolutely cannot afford to have some processes receive messages and others not



Reliable Multicast with Unreliable Multicast

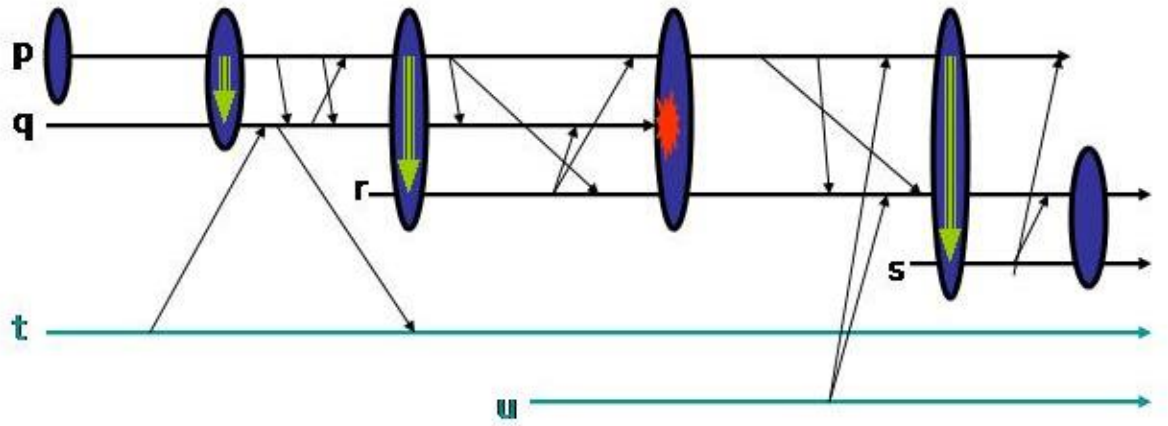
- As long as processes don't fail it's relatively easy to achieve reliable group communication via reliable multicast
- Reliable multicast can be achieved using unreliable multicast

- In the simple case sender tags messages with sequence numbers and receiver ACKs
- This leads to **feedback implosion**
- A better solution is to NACK with timeouts
  - Does require that sender retain all messages
  - Feedback implosion still occurs for larger numbers of participants
- An even better solution is to *multicast* NACKS with timeouts
  - Avoids duplicate NACKS

- How likely is feedback implosion?
- Actually pretty likely
  - WebLogic suffered from this circa 2003
  - Feedback implosion also causes collisions with other multicast messages – multicast storms

Virtual Synchrony

- Process failure is less easy to handle
- Reliable multicast in the presence of process failures can be modelled as **virtual synchrony**
- Essentially each process has a consistent **group view** of membership and the processes to which a message should be delivered
- **View changes** are then communicated and accepted interleaved with actual messages
- The key is that message delivery to the target application is not the same as receiving a message



Multicast Ordering and Atomic Multicast

- Virtual synchrony provides a synchronization primitive around group changes
  - No messages may pass the barrier
  - It means each process has a consistent view of the world
- But what about the ordering of messages? There are four possibilities

1. Unordered multicasts
2. FIFO-order multicasts - messages from the same process delivered in the order they were sent
3. Causally order multicasts - causality between messages preserved regardless of sender
4. Totally-ordered multicasts - messages delivered in the same order to all group members

- Virtual synchrony of reliable multicast with total ordering is called **atomic multicast**

Implementing Virtual Synchrony

- Messages are only declared **stable** when each process has *received* the message
- Only stable messages can be *delivered*
- Stability is ensured through a kind of 2pc protocol across group view changes

- When a view change occurs each process with unstable messages sends them to the others
- Each process then sends a flush message
- When a process has received a flush message from every other it installs the new group view

- **Isis** is an example of a system using virtual synchrony

Summary

- The possibilities for communication failure are endless
- Failures can be masked through information, time or hardware redundancy
- Reliable communication is one dimension of reliability in general
- The problem needs to be considered holistically
- The problem needs to be considered rigorously as the opportunities for mistakes are high

"We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm"

- Lamport et. al 1982

One More Thing…

- The fallacies of distributed computing
  - The network is reliable
  - Latency is zero
  - Bandwidth is infinite
  - The network is secure
  - Topology doesn't change
  - There is one administrator
  - Transport cost is zero
  - The network is homogeneous
- It happens
  - Don’t assume it doesn’t
  - All solutions are compromise …
  - … but understand the compromise
- Ideally have someone else solve the problem
  - Many products do this well

PUSH TECHNOLOGY™

Q&A

QCon

Please evaluate my talk via the mobile app!