Systèmes et Applications Répartis

Fundamentals – Part Four

Professeur Olivier Gruber

Université Joseph Fourier

Projet SARDES (INRIA et IMAG-LSR)

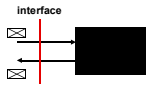
Olivier.Gruber@inria.fr

Outline

- Fault Tolerance
  - Discussing about faults and failures
  - Talking about dependability and other properties
  - Replication is at the heart of fault-tolerance for distributed systems
- Core Techniques
  - Software replication
    - Primary-based and active-replication approaches
    - The Fischer-Lynch-Paterson (FLP) impossibility result
    - Practical solutions, introducing imperfect failure detection

Definitions

- Failed System
  - *A system has failed when it does not behave according to its specification*
    - This is not a precise definition, it is system-dependent
    - This assumes that the specification is complete and correct
  - Black-box model
    - A distributed system is a collection of collaborating parts
      - Each part is considered a black-box from a failure model perspective
      - We will call each part a component
    - Failures are witnessed from outside
      - A component does not behave according to its specification
      - Example: it does not reply to messages



Definitions

- Fail-Stop Systems
  - The system stops when failing
    - It does neither send nor receive messages when failed
  - Part of the developers' job is to ensure a fail-stop behavior
    - Like proper exception catching and fail-fast philosophy
      - Detecting internal faults is difficult (always special unforeseen cases)
      - In doubts, commit suicide
        - Remember, 80% of a DBMS code base is impacted by fault-tolerance
        - So this is hard…
      - We will use replicated components to tolerate faults
  - Recovery
    - Components may recover after they failed
      - Like a reboot of a failed machine or a restart of an application
    - Requires to re-join the distributed system
      - Remember a distributed system is a collection of collaborating systems
      - Components constitute an overlay network

Definitions

- Byzantine Failures
  - The system fails but does not stop
    - It receives messages and may even send messages
    - But does not respect its specification
  - Extremely hard to detect and recover from
    - Usually corrupts the distributed state of the system
    - As well as corrupts the local state of the managed information
  - Examples
    - Calculus errors (like float overflow or loss of precision)
      - May lead a component to behave erroneously
    - Poor handling of exceptions
      - Partially corrected the situation
      - Yields incorrect behaviors at higher levels
    - Loose pointers
      - Writing in the wrong places
      - But no crash happens quickly...

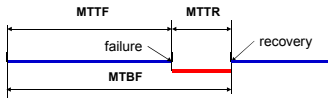
Practicality

- Practical Assumptions
  - Loss-less and FIFO channels
  - Fail-stop components
- Byzantine Failures
  - Too hard in most environments
    - Important in special environments: nuclear plants, space crafts, etc.
    - Require high redundancy
      - Typically 3k+1 replicas for resisting k failures
      - Highly based on quorum voting (the majority is correct)
  - A domain for advanced research...
    - Real important for our everyday lives as we grow dependent on distributed systems...

Definitions

- Dependability
  - Probability that the system is functioning
  - Typical measure: Mean Time To Failure (MTTF)
- Recovery
  - Time to recover from a failure
  - Typical measure: Mean Time To Recovery (MTTR)
- Availability
  - Percentage of time that the system is functioning over a period of time
  - Defined as MTTF/(MTTF+MTTR)

Definitions



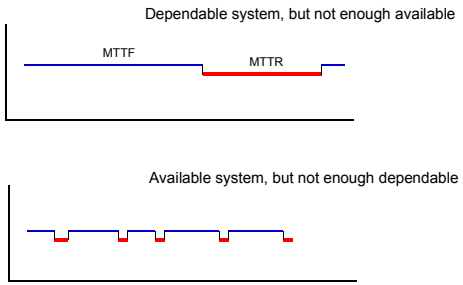
- Availability = MTTF / (MTTF + MTTR)
  - Measured in number of nines

    99.99% (4 nines) unavailable 50 minutes/year

    99.999% (5 nines) unavailable 5 minutes/year

Could mean a lot of millions of euros lost...
Or a nuclear incident or a plane crash...
Or a pissed kid that lost is on-line game...

Definitions



Olivier.Gruber@inria.fr

Fault Tolerance

- Fault masking
  - Faults are transparently recovered
    - Enough redundancy and error checking
    - Done real low in the architecture, often in hardware or in drivers
  - Example:
    - Memory parity errors and checksum recovery
    - Redundant processing units and majority vote
    - RAID disks
- Fault recovery
  - Faults do happen and software components do fail
  - To ensure good performance and long-term operation
    - Failures must be detected
    - Failures must be recovered from
  - Classical approach
    - **Fail-stop, repair, and reinsert**

Olivier.Gruber@inria.fr

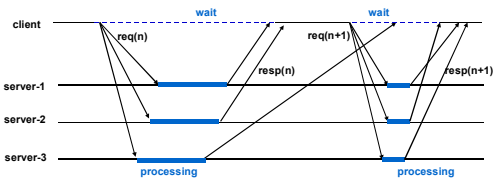
Replicated Servers

- Goal
  - High-availability servers, wanting to resist server failures
- Architecture
  - For clients
    - The model must be equivalent to a centralized server
  - Replicated servers
    - N servers resist up to **N-1 concurrent failures**
    - Failed servers are **repaired and re-inserted**
    - **Assume fail-stop servers**
  - Two models
    - Primary-based replication
    - Active replication

Olivier.Gruber@inria.fr

Active Replication

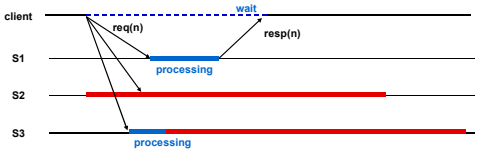
- Each client sends its requests to all servers in parallel
  - Each request has a sequence number (local for each client)
  - For each request, the client waits for the first answer, drops the following ones
- All servers are equal
  - They all process requests
  - They only update local data items



Olivier.Gruber@inria.fr

Active Replication

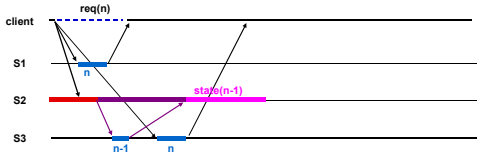
- Fault-tolerance
  - Clients need to receive at least one answer (requires at least one correct server)
    - Consider FIFO and lossless communications between clients and servers
  - Requires fail-stop servers
    - Do not send erroneous answers
  - Repair and reinsert failed servers
    - Required to preserve long-term fault-tolerance



Olivier.Gruber@inria.fr

Active Replication

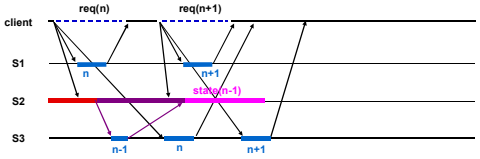
- Repair and reinsert failed servers
  - Detect failures... false-positive may happen
- Recover the state, if lost or corrupted
  - Requests it from another server
- Assert the state level
  - For each client, it will be up to a certain request-id



Olivier.Gruber@inria.fr

Active Replication

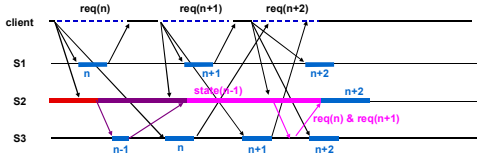
- Repair and reinsert failed servers
  - We lost all requests up to n, but we don't know it
  - We acquired state(n-1)
  - While acquiring state, we lost req(n+1)



Olivier.Gruber@inria.fr

Active Replication

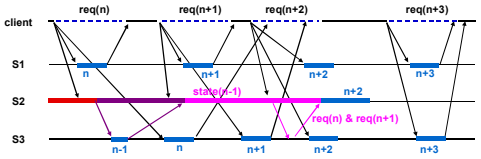
- Repair and reinsert failed servers
  - Having state(n-1), we can't process req(n+2)
    - But we now know which requests we missed: req(n+1) and req(n+2)
  - We request these missed requests from S3
    - We process them on state(n-1)
    - We are up to state(n+2) after that processing



Olivier.Gruber@inria.fr

Active Replication

- Repair and reinsert failed servers
  - Back to normal...
    - We receive req(n+3), we have state(n+2)
  - **WARNING: there can be multiple clients...**
    - So we manage vectors of sequence numbers from clients
    - So we need to totally order the requests on replicas
    - We are only back to normal when we have received all request logs that we missed



Olivier.Gruber@inria.fr

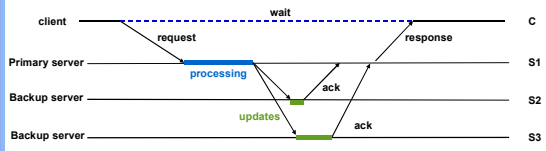
Replicated Servers

- Active Replication
  - Only works with deterministic requests
  - All servers are equivalent, they all carry the processing of client requests
  - Faster response time but costly infrastructures
  - Failures are hidden with transparent fail-over

Olivier.Gruber@inria.fr

Replicated Servers

- Primary-base Replication
  - One server is the primary, the others are backups
    - The primary executes the client requests
    - It updates locally one or more data items (x, y, ... , z)
    - Updated data items (x,y,...,z) are replicated on backup servers
  - Principle
    - Primary waits for all acknowledgements from replicas
    - All replicas (backup servers) are in the same state



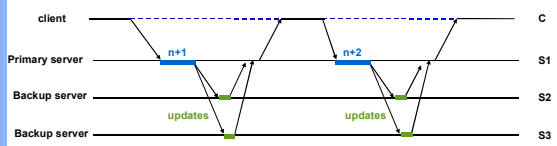
Olivier.Gruber@inria.fr

Replicated Servers

- Primary-base Replication
  - Works with non-deterministic requests
  - Require smaller investments, backup servers only handles updates
  - Slower response time, wait for all acknowledgements from backup servers
  - Clients see the primary failures, they have to switch to a new primary
  - Clients do not see failures of backup servers

Olivier.Gruber@inria.fr

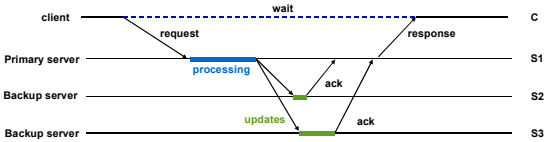
Primary-Based Replication



- Consistency Protocol (no failures)
  - Primary sets the execution order
    - Processing order of the requests
  - Communication channels
    - FIFO and loss-less
  - Clients
    - Receive only one response per request (from the primary)

Olivier.Gruber@inria.fr

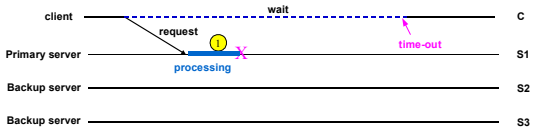
Primary-Based Replication



- Introducing Failures
  - We keep FIFO and loss-less channels
  - Both primary server and backup servers may fail
  - We consider only fail-stop servers
- Overall Goal
  - Keep all replicas consistent, despite failures

Olivier.Gruber@inria.fr

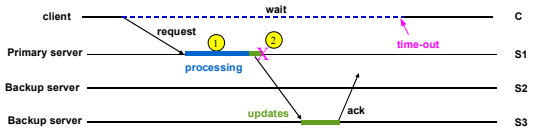
Primary-Based Replication



- Primary Failure in (1)
  - Crash happens before the processing is over
    - The client will time-out waiting for the response
    - The client will lookup the new primary and retry
  - This requires electing a new primary
    - Which requires to know the group of live servers

Olivier.Gruber@inria.fr

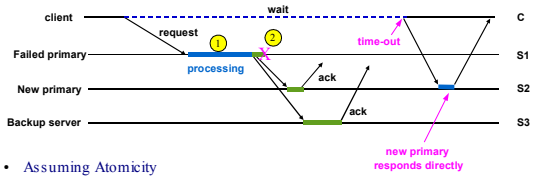
Primary-Based Replication



- Primary Failure in (2)
  - Crash happens while sending out the updates to replicas
    - The problem is that some replicas might see the updates, while others wont
  - **Atomicity has to be ensured**
    - Must get all updates or none
    - All replicas get all the updates or none of them get any update
  - If no replica received the updates
    - It is equivalent to a failure in (1)

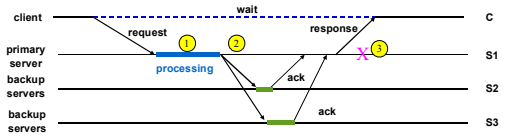
Olivier.Gruber@inria.fr

Primary-Based Replication



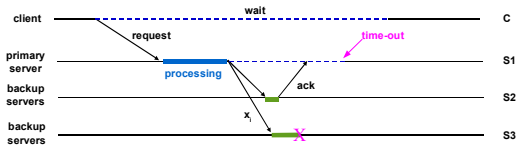
- Assuming Atomicity
  - All replicas received the updates
    - All replicas are up-to-date
    - Any replica may be elected as the new primary
  - Client will still time-out and try again
    - We must detect that the request has been processed already
    - Each request needs a unique identity (sequence number on the failed primary)
    - We need to remember the response for each request

Primary-Based Replication



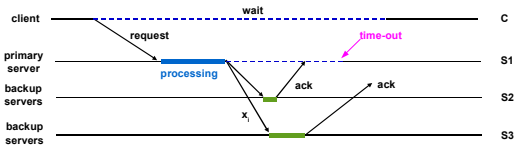
- Primary Failure in (3)
  - The client has received the response
  - It will fail to contact the primary upon its next request
    - It will time-out and lookup the newly **elected** primary
  - Eventually back to a normal situation
    - When the new primary is elected

Primary-Based Replication



- Backup Failures
  - How many acknowledgements should a primary wait for?
    - Since backups may fail, we don't know
      - We need a way to detect that a node failed
    - Synchronous system (bounded message delivery time)
      - Time-out on acknowledgement is enough
      - A time-out means the sender is down (loss-less channels)

Primary-Based Replication



- Asynchronous System
  - Message delivery is not bounded
  - **Fischer, Lynch and Paterson (FLP)**
    - In an asynchronous system with fail-stop processes
    - There is no deterministic protocol to reach a consensus

M. J. Fischer, N. Lynch, M. S. Paterson. Impossibility of Distributed Consensus with one Faulty Process, *Journal of the Association for Computing Machinery*, 32(2), pp. 374-382, April 1985 (publication initiale : *Proc. 2nd ACM Principles of Database Systems Symposium*, March 1983)

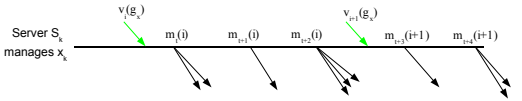
Primary-Based Replication

- Consensus Needed
  - Need to agree on which replicas are alive
  - To keep them up-to-date, to wait for acks, to elect a new primary
- Introducing View Synchronous Multicast
  - Basic idea
    - **A view is a consensus about live replicas**
    - New views are created as replicas may join, leave or fail
    - Every one in a view receives the message
  - What for?
    - So we can finish the design of primary-based replication
- Next Steps
  - Explain what is the View Synchronous Multicast
  - Explain how to use it for primary-based replication
  - Discuss consensus that is the foundation of the view mechanism

Olivier.Gruber@inria.fr

View Synchronous Multicast

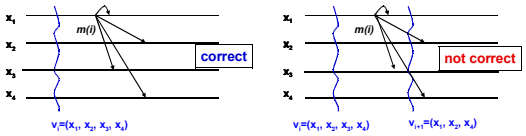
- Principles
  - Consider a group of replicas x_i, for a data item x, noted g_x
  - Consider a sequence of views $v_i(g_x)$, $v_{i+1}(g_x)$, ... $v_{i+n}(g_x)$
    - Each view represents a new state of the group
    - A new view is created everytime a node joins or leaves (includes failure)
  - Assume a node timestamps its messages with the current view
    - Let $t^k(i)$ be the local time at which replica x_k delivers the view $v_i(g_x)$
    - From $t^k(i)$, any message that x_k sends is timestamped with i, noted m(i)
    - This remains true until x_k delivers the view $v_{i+1}(g_x)$



Olivier.Gruber@inria.fr

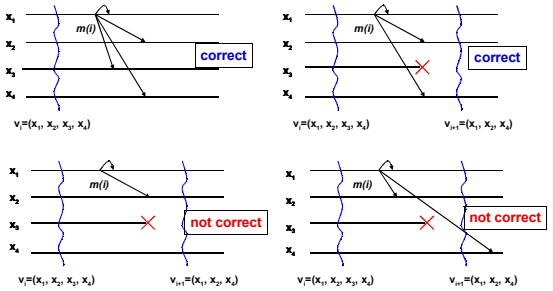
View Synchronous Multicast

- Correctness Rule
  - Given a view $v_i(g_x)$ and a message m(i)
  - All replicas in $v_i(g_x) \cap v_{i+1}(g_x)$ must either
    - all deliver m(i) before delivering $v_{i+1}(g_x)$
    - or none of them delivers m(i)



Olivier.Gruber@inria.fr

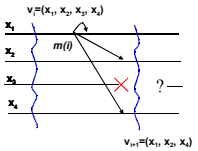
View Synchronous Multicast



Olivier.Gruber@inria.fr

Primary-Based Replication

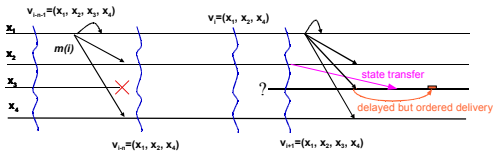
- Replica Consistency Conditions
  - If we have a failure detector (producing the views)
  - And we have a mechanism to ensure view synchronous multicasts
  - Then we have consistent replicas
- Is that enough?
  - View synchronous multicast is not enough
    - It provides reliable multicast
    - Hence atomic updates across correct replicas
  - After failure at replica x_i
    - Replica x_i is repaired and needs to re-join
    - **Its state needs to be brought up to date**



Olivier.Gruber@inria.fr

Primary-Based Replication

- State Transfers
  - To re-join, replica x_p forces a new view $v_{i+1}(g_x)$
    - Replica x_p is added $v_{i+1}(g_x)$
    - Any correct replica x_q can send its state to x_p
      - It sends its state when it delivers the new view $v_{i+1}(g_x)$
    - From the time it delivers $v_{i+1}(g_x)$
      - Replica x_p has to delay delivering all messages m(i+1)
      - Until it receives its new state from x_q



Olivier.Gruber@inria.fr

Consensus

- Definition
  - Given a set of processes $P_1,...,P_n$
  - Initially, each process P_i proposes a value V_i
  - If the consensus protocol terminates, we have
    - **Agreement**: All ***correct*** processes decide the same value
    - **Integrity**: each process decides at most once
    - **Validity**: the decided value is one of the proposed ones
    - **Decision**: if at least one correct ***process*** starts the consensus, all correct processes ***eventually*** decide a value
  - A process is correct
    - If it is not failed
    - If it has never failed (assuming a failed process may be restarted)
      - A notion that only applies within the start-end bounds of the consensus protocol

Olivier.Gruber@inria.fr

Consensus

- Starting a Consensus
  - Not included in the consensus protocol itself
    - Initially, each process P_i proposes a value V_i
  - Different possible approaches
    - It could be at regular intervals or well-know times
    - It could be by broadcasting to the processes
      - But be really careful about the properties of this broadcast
      - Only those receiving the message will be part of the consensus
- Communication Channels
  - Processes are connected through communication channels
    - Channels are FIFO and loss-less
  - We will consider synchronous and asynchronous systems
    - Delivery time is bounded or not
  - We will consider only fail-stop system
    - Byzantine failures are too complex

Olivier.Gruber@inria.fr

Consensus

- Reliable Broadcast
  - A foundation mechanism
  - A process P_i broadcast a message to all processes P_j, including itself
- Reliable Broadcast Properties
  - **Agreement:** if one correct process delivers a message, all correct processes eventually deliver m
  - **Validity**: if one correct process broadcast a message m, all correct processes eventually deliver the message
  - **Integrity**:
    - A broadcasted message is delivered at most once
    - A delivered message must have been broadcasted

Eventually: Delivery will happen in finite time

Olivier.Gruber@inria.fr

Reliable Broadcast

- Protocol

Broadcast a message, noted *m*

Timestamp m with a sequence number, noted *seq(m)*

Identify sender, noted *sender(m)*

Send m to all processes, including *sender(m)*

Deliver a broadcasted message at a process P_i

Receive the message m (from the communication channel)

If the message has been delivered, just drop it

If this is the first time P_i receive m and sender(m) is not P_i

Send m to all processes (but process P_i)

Deliver message m

Olivier.Gruber@inria.fr

Reliable Broadcast

- Discussion
  - Nothing is said about the order of delivery
  - Atomicity property
    - All correct processes eventually receive a broadcasted message
    - Or none of them receive it
- Remarks
  - It is this atomicity about a global knowledge (the message m) that allows to reason and make progress about a consensus
  - The algorithm is not optimized, better protocols exist, but the protocol shows it is possible to achieve a reliable broadcast under our assumptions

V. Hadzilacos , S. Toueg, Fault-Tolerant Broadcast and Related Problems, in S. Mullender (ed.), *Distributed Systems* (2nd edition), Addison-Wesley, 1993

Olivier.Gruber@inria.fr

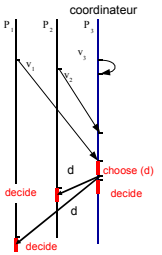
Reliable Broadcast

- Proof
  - Agreement:
    - If one process delivers a message m, it finished sending the message to all other processes prior to delivering it
    - Since communication channels are loss-less, all correct processes will eventually receive the message and deliver it (unless they crash, in which case they are not correct any more)
  - Validity:
    - If a correct process has broadcasted a message (the broadcast pseudo code was executed) the message was sent to all processes
    - Since the sender is correct (it is not failed and didn't fail), it eventually delivered the message and because of the agreement above all correct processes also delivered the message
  - Integrity
    - By the very structure of the algorithm
    - Only sent messages are received and already delivered messages are ignored

Olivier.Gruber@inria.fr

Consensus

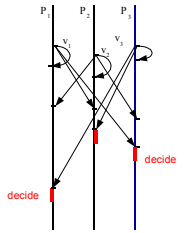
- Hypothesis
  - Loss-less communication channels
  - No node and process failures
- Coordinator Solution
  - Each process sends its value to the coordinator
  - When the coordinator has all the values, it picks one (on whatever criterium) and sends that value to all processes
  - When receiving the value, all processes decide the same value



Olivier.Gruber@inria.fr

Consensus

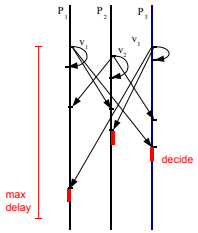
- Same Hypothesis
  - Loss-less communication channels
  - No node and process failures
- Symmetric Solution
  - All processes are equivalent
  - Each process broadcasts its initial value to all other processes
  - When a process has received all the values, it picks one using an agreed upon algorithm
  - All processes have all the same values, the same decision algorithm, they will decide the same value



Olivier.Gruber@inria.fr

Consensus

- **New** Hypothesis
  - Loss-less communication channels
  - **Fail-stop processes**
  - **Synchronous system**
- Symmetric Solution
  - Same symetric solution
  - Wait for values only for a maximum delay
  - The maximum delay can be estimated (synchronous system)
  - Passed that delay, we know that if we didn't get a message, the sender has failed
  - True only if the sending of the initial values are somewhat coordinated



Olivier.Gruber@inria.fr

Consensus

- Moving to Asynchronous Systems
  - We are facing FLP…
- Different Approaches
  - Partially synchronous systems
    - Dwork, Lynch, Stockmeyer (1988)
  - Non-deterministic algorithms
    - Rabin (1983)
  - Best-effort approaches
    - Paxos Algorithm (Lamport 1989), even adaptable to byzantine failures
    - Use imperfect fault-detectors like Chandra and Toueg (1991)

Olivier.Gruber@inria.fr

Best-Effort Consensus

- One Study Only
  - Only looking at the use of imperfect fault detectors
  - Basic idea:
    - The FLP impossibility relies on the inability to know if some process has failed or if the message we are waiting for is late, delayed in transit
    - Having a fault detector, even imperfect, is enough to avoid the FLP impossibility and make reaching a consensus possible
  - Remember:
    - Loss-less communication channels (messages will eventually arrive)
    - Asynchronous system (no bound on message delivery time)

Imperfect Failure Detectors

- Completeness
  - **Strong Completeness**: eventually, every process that crashes is permanently suspected by *every* correct process
  - **Weak Completeness**: eventually, every process that crashes is permanently suspected by *some* correct process
- Accuracy
  - **Strong Accuracy:** no process is suspected before it crashes
  - **Eventual Strong Accuracy**: eventually, correct processes are not suspected by any correct process.
  - **Weak Accuracy:** some correct process is never suspected
  - **Eventual Weak Accurary:** eventually, some *correct* process is never suspected by any *correct* process

Imperfect Failure Detectors

- Practical Choice
  - **Strong Completeness**: eventually, every process that crashes is permanently suspected by *every* correct process
  - **Eventual Weak Accurary:** eventually, some *correct* process is never suspected by any *correct* process
- Simple Design
  - Each process q periodically sends a message q-*is-alive*
    - If a process p times-out without receiving anything from q
      - It adds q to a list of suspected processes (failed)
    - If a process p realizes it erroneously suspected q
      - It removes the process q from the suspected list
      - It increments the time-out for that process q
        - Trying to safeguard against the same mistake…
- Does it work?

Imperfect Failure Detectors

- Does it work? Nope.
  - If it really did, FLP impossibility would not stand!
- But it is enough in practice…
  - As we grow the timeout
    - More and more likely that a correct process will be considered live
      - So we achieved eventual weak accuracy…
      - But no theoritical proof, just practical behavior of real systems
    - Longer will be the delay before we consider a failed process
      - So we endanger strong completeness

Strong Completeness: eventually, every process that crashes is permanently suspected by *every* correct process

Eventual Weak Accurary: eventually, some *correct* process is never suspected by any *correct* process

Consensus Protocol

- Hypothesis
  - Strong completeness and eventual weak accuracy
  - Uses a reliable broadcast noted *R-broadcast(m)*
  - Want to resist *F* failures, we need *(2F+1)* processes
- Principle
  - Tries to reach a consensus in multiple rounds
  - For each round, we try one process as the coordinator
    - If it reaches a consensus, we are done
    - If not, we try the next process as a coordinator
  - We rotate between correct processes as long as we don't have a consensus
    - Eventually, we will reach one (depending on faults and accuracy of our fault detector)
    - **No guarantee in any bounded time !**

Consensus Protocol

- Per Round
  - We have four phases
    - **Phase 1**: all processes send to the coordinator their estimate of the consensus
    - **Phase 2**: the coordinator **waits** until it has a majority of estimates, picks one as the new estimate and broadcast that new estimate
    - **Phase 3**: all processes **receive** the new estimate and acknowledge that new estimate to the coordinator
    - **Phase 4**: the coordinator **waits** for a majority of acknowledgements and then decide for that last estimate that it **reliably broadcast**
  - If anything fails to happen that way, we go for another round.
    - The coordinator may suspect a majority of processes to have failed
    - A process may suspect the coordinator to have fail and not acknowledge the new estimate
    - The coordinator may suspect a majority of processes to have failed while waiting for the acknowledgements of the last estimate

Consensus Protocol

```
upon propose(v)                                  // processus p_i
   r:=0                                          // current round
   t:=0                                          // last round where v was updated
   while not decided do
      c := (r mod N) + 1                         // p_c is the coordinator
      send (vote, r, v, t) to p_c                // N is the number of processes
                                                                          } phase 1

      if i = c then // only happens at the coordinator
         wait until (receive (vote, r, r, v', t') from (N+1)/2 non-suspected processes)
         maxt := largest t' received
         v := some v' received with t' = maxt
         send (propose, r, v) to all             // v is the new proposed consensus
                                                                          } phase 2

      wait until (receive (propose, r, v') from  p_c or c is suspected )
      if a (propose, r, v') message was received then
         v := v' ; t := r                        // Update proposed consensus
         send (ack) to p_c                       // Acknowledge proposal
      else send (nack) to p_c                    // p_i suspects the coordinator
                                                                          } phase 3

      if i = c then // only happens at the coordinator
         wait until (receive ack or nack messages from (N+1)/2 non-suspected processes)
         if all are ack then R-broadcast(decide, v)
                                                                          } phase 4
      r := r + 1                                 // try another round...

upon R-deliver (decide, v')                      // Deliver procedure of the reliable broadcast
   if not decided then
      decide(v')
      decided := true
```


Consensus Protocol

- Remarks
  - If we want to consider crash-recovery, we need a modified protocol

    M. Aguilera, W. Chen, S. Toueg. Failure detection and consensus in the crash-recovery model, *Proc 12th Int. Symp. on Distributed Computing*, 1998
  - So we achieved consensus with
    - Strong completeness
    - Eventual weak accuracy
    - We tolerate (n/2)-1 failures
    - Number of rounds is finite, but not bounded
  - Can we do better?
    - Nope, our assumptions are the weakest that solves the consensus
    - Chandra, Hadzilacos, Toueg (1996)

Conclusion

- Replication
  - We have seen two basic models (primary-based and active)
  - In synchronous and failure-free systems
    - It is rather easy
    - With fail-stop processes, it is harder
  - In asynchronous system
    - With fail-stop processes, it is complex
  - Byzantine failures are a research topic for all practical purposes
- Consensus
  - Equivalent to View Synchronous Multicast
  - Also equivalent to totally-ordered and reliable multicast
  - So both primary-based and active replication need consensus
  - Consensus is a core challenge of asynchronous distributed systems

Olivier.Gruber@inria.fr

Ariane 501

Le 4 juin 1996, le premier tir de la fusée Ariane 5 se termina par l'explosion de l'engin environ 40 secondes après le décollage.

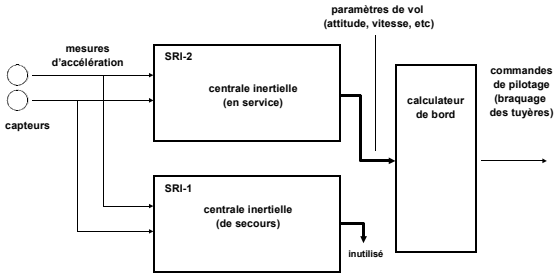
L'enquête permit de déterminer les circonstances précises de la catastrophe et mit en évidence plusieurs erreurs de conception du logiciel de commande, et notamment de son système de tolérance aux fautes, qui conduisirent au déclenchement d'une réaction inappropriée alors que les conditions de vol étaient normales.

voir rapport d'enquête en :

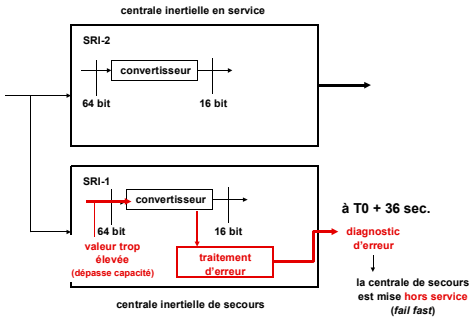
http://www.cnes.fr/espace_pro/communiques/cp96/rapport_501/rapport_501_2.html

Ariane 501: scénario (très) simplifié (1)

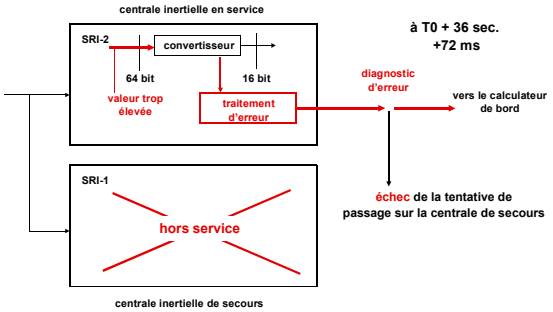
Le système de pilotage



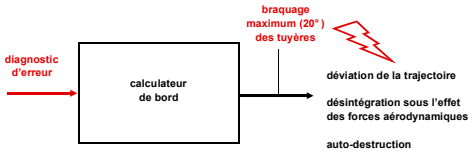
Ariane 501: scénario (très) simplifié (2)



Ariane 501: scénario (très) simplifié (3)



Ariane 501: scénario (très) simplifié (4)



Le calculateur de bord reçoit en entrée un diagnostic d'erreur, qu'il interprète comme des données de vol. Les valeurs étant hors limites, il réagit (normalement) par un ordre de braquage maximum

Ariane 501 : explications (1)

Pourquoi le convertisseur a-t-il reçu en entrée une valeur trop grande ?

1) L'entrée était la mesure d'une vitesse de déplacement horizontale, et servait à un système de calibrage des instruments utilisé avant le décollage. Ce système était inchangé depuis sa version Ariane 4 (on ne change pas un dispositif qui marche…).
2) Néanmoins, ce système était maintenu en service 40 secondes après le décollage, en vue de permettre l'arrêt puis la reprise du compte à rebours dans les dernières secondes avant le décollage (dans une limite de 40 secondes). Cette disposition (qui a servi une fois pour Ariane 4) était en fait inutile pour Ariane 5 en raison d'un changement des procédures.
3) Mais la trajectoire d'Ariane 5 était différente de celle d'Ariane 4, et en particulier sa vitesse de déplacement horizontale était beaucoup plus élevée
4) Donc la valeur de l'entrée, correspondant à la vitesse horizontale mesurée, a dépassé les capacités du convertisseur

Ariane 501 : explications (2)

Pourquoi le convertisseur a-t-il réagi en envoyant un diagnostic d'erreur sur le bus de données ?

1) Les instructions de conversion de données (écrites en Ada) n'étaient pas protégées par un traitement d'exception (on n'avait pas prévu que la capacité d'entrée pouvait être dépassée).

2) C'est le dispositif standard de réaction aux erreurs qui a été activé par défaut. La centrale à inertie a déclaré son incapacité à fonctionner et a été mise hors service selon le principe *fail stop* (arrêt du processeur).

3) L'hypothèse sous-jacente (non explicite) était celle de la défaillance matérielle transitoire : défaillance de probabilité très faible, traitée par redondance (on passe sur la centrale de secours).

4) Mais la défaillance étant d'origine logicielle, les même causes ont produit des effets identiques, et la centrale de secours s'est trouvée hors service pour les mêmes raisons que la centrale active.

5) On se trouvait donc hors des hypothèses prévues pour le traitement des défaillances (panne simultanée des deux unités). Ce cas étant supposé ne jamais se présenter, rien n'était prévu pour le traiter et le diagnostic d'erreur a été envoyé sur le bus de données.

Ariane 501 : la morale de l'histoire

Le scénario de la catastrophe révèle plusieurs erreurs de conception

1) Analyse incorrecte de la transition Ariane 4 → Ariane 5 (pour la centrale à inertie)
   - Un dispositif totalement inutile a été maintenu en fonction dans un équipement critique
   - Les conséquences du changement de conditions (vitesse différente) n'ont pas été analysées

2) Analyse incorrecte des hypothèses de défaillance
   - Le dépassement de capacité n'a pas été prévu dans le convertisseur (pas de traitement d'exception associé à l'opération de conversion).
   - L'hypothèse à la base de la redondance a été incorrectement formulée (la redondance ne vaut que si les conditions de défaillance sont indépendantes pour les deux unités redondantes).
   - L'éventualité d'erreurs logicielles (qui précisément ne sont pas indépendantes, surtout si un même algorithme sert dans les deux unités) n'a pas été prise en compte.

3) Prévision insuffisante des conditions de traitement de situations anormales
   - La réaction de mise hors service de la centrale après erreur (arrêt du processeur) prive le système d'une possibilité de reprise (cette erreur d'analyse est liée à l'idée que la défaillance est toujours traitée par la redondance des unités).
   - La possibilité d'envoyer des informations de diagnostic sur un bus de données aurait du être totalement exclue. Il valait mieux, en cas de situation anormale, envoyer des données "raisonnables", par exemple obtenues par extrapolation.