The University *of* New South Wales School *of* Computer Science & Engineering

COMP9243 — Lecture 8 (20T3)

Ihor Kuz, Gernot Heiser

Fault Tolerance

In previous lectures we've mentioned that one of the reasons that distributed systems are different (and more complicated) than nondistributed systems is due to partial failure of system components. We've mentioned that dependability is an important challenge in designing and building distributed systems and that the presence of failure often makes achieving transparency (e.g., for RPC) difficult if not impossible. In this lecture we take a deeper look into the concept of failure and how failure is dealt with in distributed systems.

Dependability

Dependability is the ability to avoid service failures that are more frequent or severe than desired. A key requirement of most systems is to provide some level of dependability. A dependable systems has the following properties.

Availability: refers to the probability that the system is operating correctly at any given moment and is available to perform its functions. Availability is often given as a percentage, e.g., 99.9999% availability means that 99.9999% of the time the system will be operating correctly. Over a year this amounts to less than a minute of downtime.

Reliability: refers to the ability of the system to run continuously without failure. Note that this is different than availability. A 99.9999% available system could go down for a millisecond every hour. Although the availability would be high, the reliability would be low.

Safety: refers to the fact that when a system (temporarily) fails to operate correctly, nothing catastrophic happens. For example, the controller of a nuclear power plant requires a high degree of safety.

Maintainability: refers to how easily a failed system can be repaired. This is especially useful if automatic recovery from failure is desired and can lead to high availability.

Integrity, Confidentiality : are related to security and will be dealt with in a separate lecture.

Building a dependable system comes down to preventing failure.

Faults and Failures

In the following discussion of failure and fault tolerance we use the following terminology:

A system is a set of hardware and software components designed to provide a specific service. Its components may also be (sub)systems.

A failure of a system occurs when the system fails to meet its promises or does not perform its services in the specified manner. Note that this implies a specification or understanding of what correct services are.

An erroneous state is a state which could lead to a system failure by a sequence of valid state transitions.

An error is a part of the system state which differs from its intended value. An error is a manifestation of a fault in the system, which could lead to system failure.

A fault is an anomalous condition. Faults can result from design errors, manufacturing faults, deterioration, or external disturbance.

Failure recovery is the process of restoring an erroneous state to a error-free state.

A good overview of the relation between faults, errors, and failures can be found in [ALRL04].

Faults

Failures are caused by faults. In order to better understand failures, and to better understand how to prevent faults from leading to failure we look at the different properties of faults. We distinguish between three categories of faults.

Transient faults are those that occur once and never reoccur. These are typically caused by external disturbances, for example, wireless communication being interrupted by external interference, such as a bird flying through a microwave transmission. *Intermittent faults* are those that reoccur irregularly. Typically these kinds of faults occur repeatedly, then vanish for while, then reoccur again. They are often caused by loose contacts, or non-deterministic software bugs such as race conditions. Finally *permanent faults* are those that persist until the faulty component is replaced. Typical examples of permanent faults are software bugs, burnt out chips, crashed disks, etc.

Faults are typically dormant until they are activated by some event causing them to create an error.

When systems rely on other (sub)systems, faults can propagate, causing failures in multiple systems. For example, a fault in a disk can cause the disk to fail by returning incorrect data. A service using the disk sees the disk failure as a fault that causes it to read an incorrect value (i.e., an error), which in turn causes it to provide an incorrect service (e.g., return an incorrect reply to a database query).

Failures

There are different kinds of failures that can occur in a distributed system. The important types of failure are:

System failure: the processor may fail to execute due to software faults (aka. an OS bug) or hardware faults (affecting the CPU, main memory, the bus, the power supply, a network interface, etc.) Recovery from such failures generally involves stopping and restarting the system. While this may help with intermittent faults, or failures triggered by a specific and rare combination of circumstances, in other cases it may lead to the system failing again at the same point. In such a case, recovery requires external interference (replacing the faulty component) or an automatic reconfiguration of the system to exclude the faulty component (e.g. reconfiguring the distributed computation to continue without a faulty node).

Process failure: a process proceeds incorrectly (owing to a bug or consistency violation) or not at all (due to deadlock, livelock or an exception). Recovery from process failure involves aborting or restarting the process.

Storage failure: some (supposedly) stable storage has become inaccessible (typically a result of hardware failure). Recovery involves rebuilding the device's data from archives, logs, or mirrors (RAID).

Communication failure: communication fails due to a failure of a communications link or intermediate node. This leads to corruption or loss of messages and may lead to partitioning of the network. Recovery from communication medium failure is difficult and sometimes impossible.

In nondistributed systems a failure is almost always *total*, which means that all parts of the system fail. For example, if the operating system that an application runs on crashes, then the whole application goes down with the operating system. In a distributed system this is rarely the case. It is more common that only a part of the system fails, while the rest of the system components continue to function normally. For example, in a distributed shared memory system, the network link between one of the servers and the rest of the servers may fail. Despite the network link failing the rest of the components (all the processes, and the rest of the network links) continue to work correctly. This is called *partial failure*. In this lecture we concentrate on the problems caused by partial failures in distributed systems.

There are a number of different ways that a component in a distributed system can fail. The way in which a component fails often determines how difficult it is to deal with that failure and to recover from it.

Crash Failure: a server halts, but works correctly until it halts

> **Fail-Stop:** server will stop in a way that clients can tell that it has halted.
>
> **Fail-Resume** server will stop, then resume execution at a later time.
>
> **Fail-Silent:** clients do not know server has halted

Omission Failure: a server fails to respond to incoming requests

- *Receive Omission:* fails to receive incoming messages
- *Send Omission:* fails to send messages

Timing Failure: a server's response lies outside the specified time interval

Response Failure: a server's response is incorrect

- *Value Failure:* the value of the response is wrong
- *State Transition Failure:* the server deviates from the correct flow of control

Arbitrary Failure: a server may produce arbitrary response at arbitrary times (aka *Byzantine failure*)

Fault Tolerance

The topic of *fault tolerance* is concerned with being able to provide correct services, even in the presence of faults. This includes preventing faults and failures from affecting other components of the system, automatically recovering from partial failures, and doing so without seriously affecting performance.

Failure Masking

One approach to dealing with failure is to hide occurrence of failures from other processes. The most common approach to such *failure masking* is *redundancy*. Redundancy involves duplicating components of the system so that if one fails, the system can continue operating using the nonfailed copies. There are actually three types of redundancy that can be used. *Information redundancy* involves including extra information in data so that if some of that data is lost, or modified, the original can be recovered. *Time redundancy* allows actions to be performed multiple times if necessary. Time redundancy is useful for recovering from transient and intermittent faults. Finally *physical redundancy* involves replicating resources (such as processes, data, etc.) or physical components of the system.

Process Resilience

Protection against the failure of processes is generally achieved by creating groups of replicated processes. Since all the processes perform the same operations on the same data, the failure of some of these processes can be detected and resolved by simply removing them from the group, continuing execution with only the nonfailed processes. Such an approach requires the group membership to be dynamic, and relies on the presence of mechanisms for managing the groups and group membership. The group is generally transparent to its users, that is, the whole group is dealt with as a single process.

Process groups can be implemented hierarchically or as a flat space. In a hierarchical group a coordinator makes all decisions and the other members follow the coordinators orders. In a flat group the processes must make decisions collectively. A benefit of the flat group approach is that there is no single point of failure, however the decision making process is more complicated. In a hierarchical group, on the other hand, the decision making process is much simpler, however the coordinator forms a single point of failure.

Process groups are typically modelled as replicated state machines [Sch90]. In this model every replica process that is part of the group implements an identical state machine. The state machine transitions to a new state whenever it receives a message as input, and it may send one or more messages as part of the state transition. Since the state machines are identical, any process executing a given state will react to input messages in the same way. In order for the replicas to remain consistent they must perform the same state transitions in the same order. This can be achieved by ensuring that each replica receives and processes input messages in exactly the same order: the replicas must all agree on the same order of message delivery, which requires *consensus*. The result is a deterministic set of replica processes, and all correct replicas are assured to produce exactly the same output.

A group of replicated processes is k fault tolerant if it can survive k faults and still meet its specifications. With fail-stop semantics, $k+1$ replicas are sufficient to survive k faults, however with arbitrary failure semantics, $2k+1$ processes are required. This is because a majority $(k+1)$ of the processes must provide the correct results even if the k failing processes all manage to provide the same incorrect results.

Consensus

As we have seen previously, consensus (or agreement) is often needed in distributed systems. For example, processes may need to agree on a coordinator, or they may need to agree on whether to commit a transaction or not. The consensus algorithms we've looked at previously all assumed that there were no faults: no faulty communication and no faulty processes. In the presence of faults we must ensure that all nonfaulty processes reach and establish consensus within a finite number of steps.

A correct consensus algorithm will have the following properties:

Agreement all processes decide on the same value

Validity the decided value was proposed by one of the processes

Termination all processes eventually decide.

We will look at the problem of reaching consensus in synchronous and asynchronous systems. We start with synchronous systems which assume that execution time is bounded (i.e., processes can execute in rounds), and that communication delay is bounded (i.e., timeouts can be used to detect failure).

We first look at the problem of reaching consensus with nonfaulty processes, but unreliable communication. The difficulty of agreement in this situation is illustrated by the *two-army problem* (Figure 1). In this problem the two blue armies must agree to attack simultaneously in order to defeat the green army. Blue army 1 plans an attack at dawn and informs blue army 2. Blue army 2 replies to blue army 1 acknowledging the plans. Both armies now know that the plan is to attack

at dawn. However blue army 2 does not know whether blue army 1 received its acknowledgment. It reasons that if blue army 1 did not receive the acknowledgment, it will not know whether blue army 2 received the original message and will therefore not be willing to attack. Blue army 1 knows that blue army 2 may be thinking this, and therefore decides to acknowledge blue army 2's acknowledgment. Of course blue army 1 does not know whether blue army 2 received the acknowledgment of the acknowledgment, so blue army 2 has to return an acknowledgment of the acknowledgment of the acknowledgment. This process can continue indefinitely without both sides ever reaching consensus.

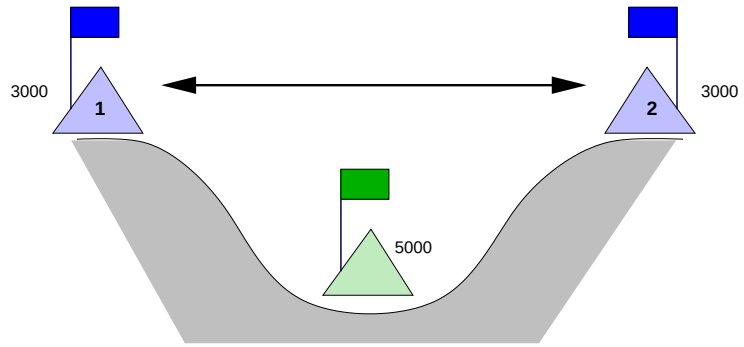


Figure 1: The two army problem

A different problem involves reliable communication, but faulty (byzantine) processes. This problem is illustrated by the byzantine generals problem, where a group of generals must come to consensus about each others troop strengths. The problem is that some of the generals are traitors and will lie in order to prevent agreement. Lamport et al. devised a recursive algorithm to solve this problem. It was also proven that in a system with k faulty (byzantine) processes, agreement can only be achieved if there are $2k + 1$ nonfaulty processes [LSP82].

In an asynchronous system the temporal assumptions are dropped: execution is no longer bounded, and communication delay is not bounded (so timeouts cannot be used to detect failure). It turns out that in an asynchronous distributed system it is impossible to create a totally correct consensus algorithm that can tolerate the failure of even one process. What this means is that no algorithm can guarantee correctness in every scenario if one or more processes fail. This was proven by Fischer, Lynch, and Patterson [FLP85] in 1985. In practice however we can get algorithms that are 'good enough'.

We revisit 2PC to see how it behaves when faults are possible. In a distributed system, either a host or the network can fail. For communication failures 2PC can use timeouts to detect such failures and recover from them. Figures 2 and 3 depict extended state transition diagrams for the coordinator and worker, respectively, that explicitly take timeouts into account. Whenever a worker that is ready to commit when receiving a `CanCommit` message times out while waiting for the decision from the coordinator, it sends a special `GetDecision` message that triggers the coordinator to resend the decision on whether to `Commit` or `Abort`. These messages are handled by the coordinator once the decision between committing and aborting has been made. Moreover, the coordinator resends `CanCommit` messages if a worker does not issue a timely reply.

In the case of host failures, if the coordinator fails after having initiated the protocol with some of the workers, these workers are blocked, since they cannot continue without further input from the coordinator. This problem can be resolved if another host takes over the coordinator's role and resumes the protocol. This recovery coordinator host can try to learn the state of the protocol from the other workers. However, if one of the workers also crashes, then it may be impossible to recover, and the protocol will be indefinitely blocked. The reason for this is that the new coordinator cannot distinguish between a scenario where all workers voted to commit and the

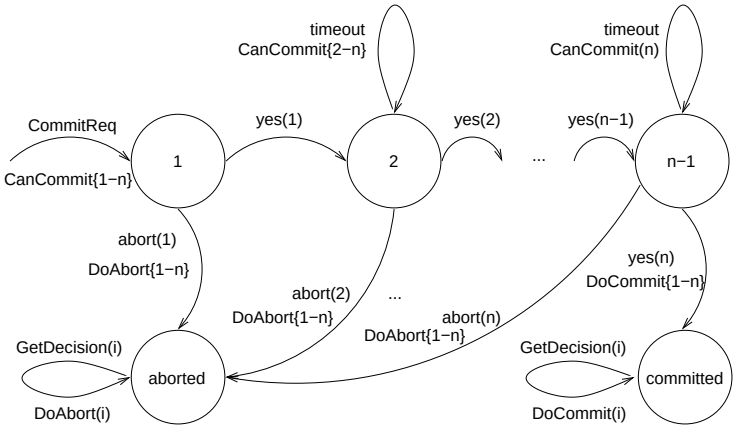


Figure 2: State transition diagram of 2PC coordinator with timeouts

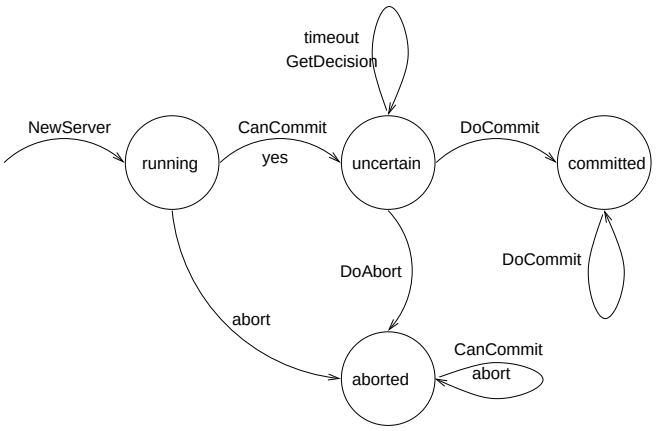


Figure 3: State transition diagram of 2PC worker with timeouts

failed node has already committed, and a scenario where the failed node voted to abort and the rest of the workers voted to commit. In the first case the recovery coordinator cannot choose to abort, while in the second case it cannot choose to commit.

A solution to this problem is to add a third phase to the protocol, resulting in *three-phase commit* (3PC) [Ske81]. In this protocol the second phase of 2PC is split into a *pre-commit* and *commit* phase. In the pre-commit phase the coordinator informs the workers of the vote results (and waits for their confirmation), and in the commit phase it requests them to perform the actual operation. The goal of the pre-commit phase is to inform all the workers of the vote, without requiring the workers to irreversibly change their state. In case of a worker failure in this phase, the whole protocol can be safely aborted. By the time the protocol enters the commit phase, the coordinator and all workers know the result of the vote, thus a worker failure can no longer leave a recovery coordinator in a confused state, and the protocol will not block. While 3PC is an improvement it still has problems in the face of network partitions and is not used much in practice.

The Paxos consensus algorithm [Lam98, Lam01] is currently the best algorithm for consensus in real distributed systems, and is used to implement highly scalable and reliable services such as Google's Chubby lock server. The goal of Paxos is for a group of processes to agree on a value, and make that value known to any interested parties. The algorithm requires a leader process to be elected and proceeds in two phases. In the first phase, the leader (called the Proposer) sends a *proposal* to all the other processes (called Acceptors). A proposal contains a monotonically increasing sequence number and a proposed value. Each acceptor receives the proposal and decides on whether to accept the proposed value or not. The proposal is rejected if the sequence number is lower than the sequence number of any value the acceptor has already accepted in the past. Otherwise, the proposal is accepted, and the acceptor sends a *promise* message, containing the value of that acceptor's most recently accepted value (if any – otherwise no value is returned in the message).

In the second phase, the proposer waits until it has received a reply from a majority of the acceptors. Given all the replies, the proposer chooses a value to propose as follows: if any of the promises contained a value, then the proposer must choose the highest of these values (that is, the value associated with the highest sequence number), otherwise it is free to choose an arbitrary value. The proposer sends its chosen value in an *accept* message to all the acceptors. Upon receiving this message each acceptor checks whether it can still accept the value. It will accept the value unless it has subsequently sent a promise with a higher sequence number to another proposer. The acceptor replies to the proposer with an *accepted* message. Once the proposer has received accepted messages from a majority of the acceptors, it knows that the proposed value has been agreed upon and can pass this on to any other interested parties.

Paxos can tolerate the failure of up to half of the acceptor processes as well as the failure of the proposer process (when a proposer fails, a new one can be elected and can continue the algorithm where the old one left off). If multiple proposers are simultaneously active, it is possible for Paxos to enter a state known as dueling proposers which could continue without progress indefinitely. In this way Paxos chooses to sacrifice termination rather than agreement in the face of failure.

Implementing this algorithm in real systems turns out to be more difficult than the simple description of it in literature would imply. An interesting account of the challenges faced while implementing a version of Paxos is given by Chandra *et al.* [CGR07].

Reliable Communication

Besides processes the other part of a distributed system that can fail is the communication channel. Masking failure of the communication channel leads to *reliable communication.*

Reliable Point-to-Point Communication

Reliable point-to-point communication is generally provided by reliable protocols such as TCP/IP. TCP/IP masks omission failures but not crash failures. When communication is not reliable, and

in the presence of crash failures, it is useful to define failure semantics for communication protocols. An example is the semantics of RPC in the presence of failures. There are five different classes of failures that can occur in an RPC system.

- Client cannot locate server
- Request message to server is lost
- Server crashes after receiving a request
- Reply message from server is lost
- Client crashes after sending a request

In the first case the RPC system must inform the caller of the failure. Although this weakens the transparency of the RPC it is the only possible solution. In the second case the sender can simply resend the message after a timeout. In the third case there are two possibilities. First, the request had already been carried out before the server crashed, in which case the client cannot retransmit the request message and must report the failure to the user. Second, the request was not carried out before the server crashed, in which case the client can simply retransmit the request. The problem is that the client cannot distinguish between the two possibilities. Depending on how this problem is solved results in *at-least-once*, *at-most-once*, and *maybe* semantics.

In the fourth case it would be sufficient for the server to simply resend the reply. However, the server does not know whether the client received its reply or not and the client cannot tell whether the server has crashed or whether the reply was simply lost (or if the original request was lost). For *idempotent* operations (i.e., operations that can be safely repeated) the client simply resends its request. For nonidempotent operations, however, the server must be able to distinguish a retransmitted request from an original request. One approach is to add sequence numbers to the requests, or to include a bit that distinguishes an original request from a retransmission.

Finally, in the fifth case, when a client crashes after sending a request, the server may be left performing unnecessary work because the client is not around to receive the results. Such a computation is called an *orphan*. There are four ways of dealing with orphans. *Extermination* involves the client explicitly killing off the orphans when the it comes back up. *Reincarnation* involves the client informing all servers that it has restarted, and leaving it up to the servers to kill off any computations that were running on behalf of that client. *Gentle reincarnation* is similar to reincarnation, except that servers only kill off computations whose parents cannot be contacted. Finally, in *expiration* each RPC is given a fixed amount of time to complete, if it cannot complete it must explicitly ask the client for more time. If the client crashes and reboots, it must simply wait long enough for all previous computations to have expired.

Reliable Group Communication

Reliable group communication, that is, guaranteeing that messages are delivered to all processes in a group, is particularly important when process groups are used to increase process resilience. We distinguish between reliable group communication in the presence of faulty processes and reliable group communication in the presence of nonfaulty processes. In the first case the communication succeeds only when all nonfaulty group members receive the messages. The difficult part is agreeing on who is a member of the group before sending the message. In the second case it is simply a question of delivering the messages to all group members.

Figure 4 shows a basic approach to reliable multicasting assuming nonfaulty processes. In this example the sender assigns a sequence number to each message and stores sent messages in a history buffer. Receivers keep track of the sequence number of the last messages they have seen. When a receiver successfully receives a message that it was expecting, it returns an acknowledgment to the sender. When a receiver receives a message that it was not expecting (e.g., because it was expecting an older message first) it informs the sender which messages it has not yet received. The sender can then retransmit the messages to that particular receiver.

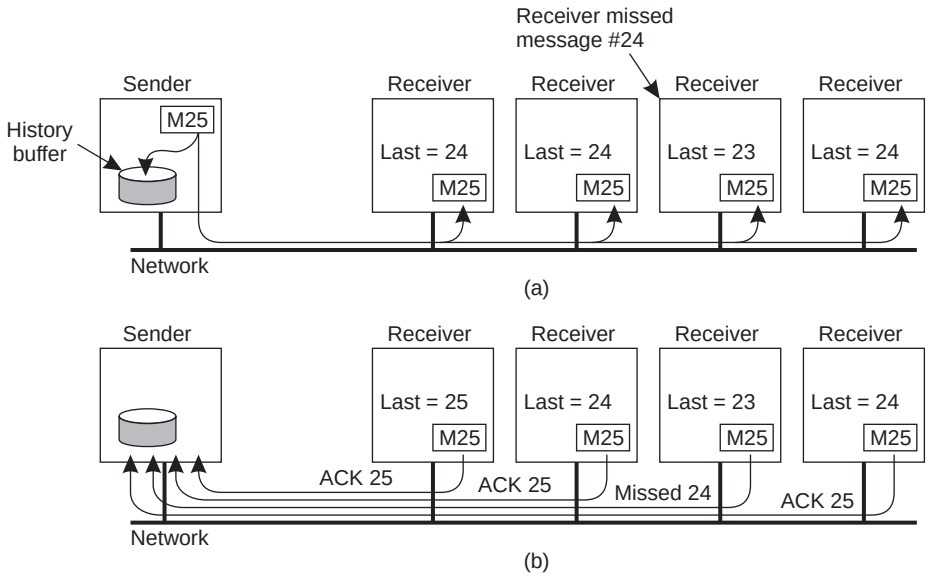


Figure 4: Basic reliable multicast approach

A problem with this approach is that if the group is large enough the sender must deal with a large amount of acknowledgment messages, which is bad for scalability. This problem is known as *feedback implosion*. In order to avoid this the multicast approach can be modified to reduce the amount of feedback the server must process. In this approach receivers do not send acknowledgments, but only negative acknowledgments (NACKs) when they are missing messages. A major drawback of this approach is that the sender must keep its history buffer indefinitely as it does not know when all receivers have successfully received a message.

A different approach to improving scalability is to arrange groups in a hierarchical fashion (see Figure 5). In this approach a large group of receivers is partitioned into subgroups, and the subgroups are organised into a tree. Each subgroup is small enough so that any of the above mentioned reliable group communication schemes can be applied and for each subgroup a local coordinator acts as the sender. All the coordinators and the original sender also form a group and use one of the above mentioned multicast schemes. The main problem with this approach is constructing the tree. It is particularly difficult to support dynamic addition and removal of receivers.

When discussing reliable group communication in the face of possibly faulty processes, it is useful to look at the *atomic multicast* problem. Atomic multicast guarantees that a message will be delivered to all members of a group, or to none at all. It is generally also required that the messages are delivered in the same order at all receivers. When combined with faulty processes, atomic multicast requires that these processes be removed from the group, leaving only nonfaulty processes in receiver groups. All processes must agree on the *group view*, that is, the view of the group the sender had when the message was sent.

Failure Recovery

The term *recovery* refers to the process of restoring a (failed) system to a normal state of operation. Recovery can apply to the complete system (involving rebooting a failed computer) or to a particular application (involving restarting of failed process(es)).

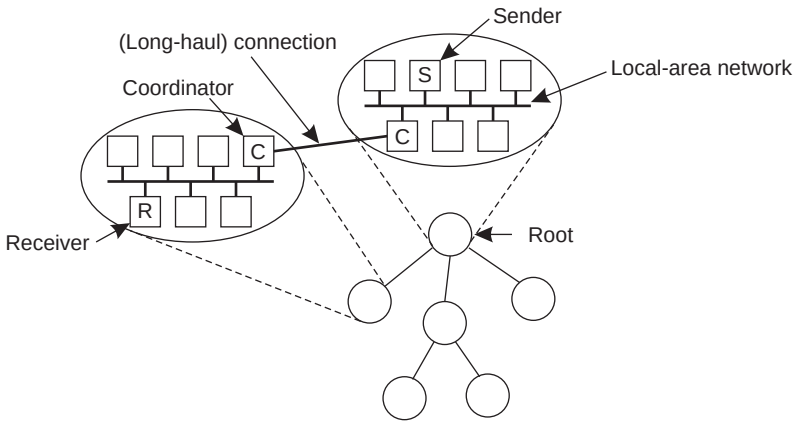


Figure 5: Hierarchical multicast

While restarting processes or computers is a relatively straightforward exercise in a centralised system, things are (as usual) significantly more complicated in a distributed system. The main challenges are:

Reclamation of resources: a process may hold resources, such as locks or buffers, on a remote node. Naively restarting the process or its host will lead to resource leaks and possibly deadlocks.

Consistency: Naively restarting one part of a distributed computation will lead to a local state that is inconsistent with the rest of the computation. In order to achieve consistency it is, in general, necessary to undo partially completed operations on other nodes prior to restarting.

Efficiency: One way to avoid the above problems would be to restart the complete computation whenever one part fails. However, this is obviously very inefficient, as a significant amount of work may be discarded unnecessarily.

Forward vs. backward recovery

Recovery can proceed either forward or backward. *Forward error recovery* requires removing (repairing) all errors in the system's state, thus enabling the processes or system to proceed. No actual computation is lost (although the repair process itself may be time consuming). Forward recovery implies the ability to completely assess the nature of all errors and damages resulting from the faults that lead to failure. An example could be the replacement of a broken network cable with a functional one. Here it is known that all communication has been lost, and if appropriate protocols are used (which, for example, buffer all outgoing messages) a forward recovery may be possible (e.g. by resending all buffered messages). In most cases, however, forward recovery is impossible.

The alternative is *backward error recovery*. This restores the process or system state to a previous state known to be free from errors, from which the system can proceed (and initially retrace its previous steps). Obviously this incurs overheads due to the lost computation and the work required to restore the state. Also, there is in general no guarantee that the same error will not reoccur (e.g. if the failure resulted from a software bug). Furthermore, there may be irrecoverable components, such as external input (from humans) or irrevocable outputs (e.g. cash dispensed from an ATM).

While the implementation of backward recovery faces substantial difficulties, it is in practice the only way to go, due to the impossibility of forward-recovery from most errors. For the remainder of this lecture we will, therefore only look at backward recovery.

Backward recovery

Backward error recovery works by restoring processes to a *recovery point*, which represents a pre-failure state of the process. A system can be recovered by restoring all its active processes to their recovery points. Recovery can happen in one of two ways:

Operation-based recovery keeps a *log* (or *audit trail)* of all state-changing operations. The recovery point is reached from the present state by reversing these operations;

State-based recovery stores a complete prior process state (called a *checkpoint).* The recovery point is reached by restoring the process state from the checkpoint (called *roll-back*). State-based recovery is also frequently called *rollback-recovery.*

Both approaches require the recovery data (log or checkpoint) to be recorded on *stable* storage. Combinations of both are possible, e.g. by using checkpoints in an operation-based scheme to reduce the rollback overhead.

Operation-based recovery is usually implemented by *in-place updates* in combination with *write-ahead logging. Before* a change is made to data, a record is written to the log, completely describing the change. This includes an identification (name) of the affected object, the pre-update object state (for undoing the operation, roll-back) and the post-update object state (for redoing the operation, roll-forward).[1] The implied *transaction semantics* makes this scheme attractive for databases.

State-based recovery requires checkpoints to be performed during execution. There exists an obvious trade-off for the frequency of checkpointing: checkpoints slow execution and the overhead of frequent checkpoints may be prohibitive. However, a low checkpoint frequency increases the average recovery cost (in terms of lost computation).

Checkpointing overhead can be reduced by some standard techniques:

Incremental checkpointing: rather than including the complete process state in a checkpoint, include only the changes since the previous checkpoint. This can be implemented by the standard memory-management technique of *copy-on-write*: After a checkpoint the whole address space is write protected. Any write will then cause a protection fault, which is handled by copying the affected page to a buffer, and un-protecting the page. On a checkpoint only those "dirty" pages are written to stable storage.

The drawback of incremental checkpointing is increased restart overhead, as first the process must be restored to the last complete checkpoint (or its initial state), and then all incremental checkpoints must be applied in order. Much of that work is in fact redundant, as processes tend to dirty the same pages over and over. For the same reason, the sum of the incremental checkpoints can soon exceed the size of a complete checkpoint. Therefore, incremental checkpointing schemes will occasionally perform a complete checkpoint to reduce the consumption of stable storage and the restart overhead.

Asynchronous checkpointing: rather than blocking the process for the complete duration of a checkpoint, copy-on-write techniques are used to protect the checkpoint state from modification while the checkpoint is written to stable storage concurrently with the process continuing execution. Inevitably, the process will attempt to modify some pages before they have been written, in which case the process must be blocked while those pages are written. Blocking is minimised by prioritising writes of such pages. The scheme works best if the checkpoint is written in two stages: first to a buffer in memory, and from there to disk. The process will not have to block once the in-memory checkpoint is completed.

[1] Strict operation-based recovery does not require the new object state to be logged, but recovery can be sped up if this information is available in the log.

Asynchronous checkpointing can be easily (and beneficially) combined with incremental checkpointing. Its drawback is that a failure may occur before the complete checkpoint is written to stable storage, in which case the checkpoint is useless and recovery must use an earlier checkpoint. Hence this scheme generally requires several checkpoints to be kept on stable storage. Two checkpoints are sufficient, provided that a new checkpoint is not commenced until the previous one is completed. Under adverse conditions this can lead to excessive delays between checkpoints, and may force synchronisation of a checkpoint in order to avoid delaying the next one further.

Compressed checkpoints: The checkpoint on stable storage can be compressed in order to reduce storage consumption, at the expense of increased checkpointing and restart costs.

These techniques can be implemented by the operating system, or transparently at user level (for individual processes)[PBKL95]. This is helped significantly by the semantics of the Unix `fork()` system call: in order to perform a checkpoint, the process forks an identical copy of itself. The parent can then continue immediately, while the child performs the checkpoint by dumping its address space (or the dirty parts of it).

Problems with recovery

The main challenge for implementing recovery in a distributed system arises from the requirement of consistency. An isolated process can easily be recovered by restoring it to a checkpoint. A distributed computation, however, consists of several processes running on different nodes. If one of them fails, it may have *causally affected* another process running on another node. A process B is causally affected by process A if any of A's state has been revealed to B (and B may have subsequently based its computation on this knowledge of A's state). See *Global States* for details.

Domino rollback If A has causally affected B since A's last checkpoint, and A subsequently fails and is restored to its checkpointed state, then B's state is *inconsistent* with A's state, as B depends on a possible *future* state of A. It is possible that in its new run A does not even enter that state again. In any case, the global state may no longer be consistent, as it may not be reachable from the initial state by a fault-free execution.

Such a situation must be avoided, by rolling back all processes which have been causally affected by the failed process. This means that *all* processes must establish recovery points, and that furthermore any recovery must be guaranteed to roll back to a *consistent global state*. The problem is that this can lead to *domino rollback*, as shown in Figure 6.

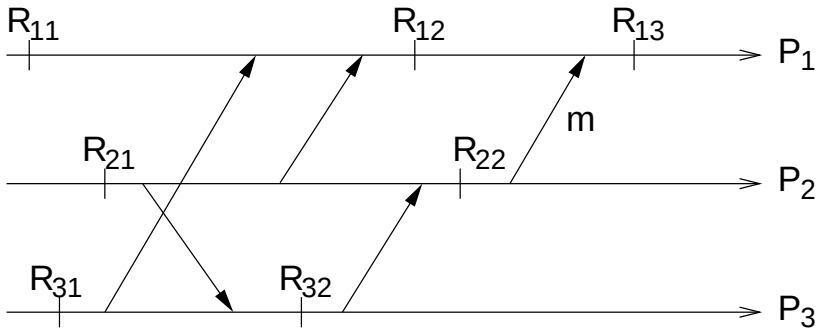


Figure 6: Domino effect leads to total rollback after failure of P_3

If P_1 fails, it can be rolled back to R_{13} which leaves the system in a consistent state. We use the notation $P_1\Downarrow$ to indicate that P_1 fails, and $P_1 \curvearrowright R_{13}$ to indicate P_1 being rolled back to R_{13}.

A failure of P_2 is more serious: $P_2\Downarrow \Rightarrow P_2 \curvearrowright R_{22}$ leads to an inconsistent state, as the state recorded in R_{22} has causally affected the state recorded in R_{13}, and hence P_1's present state.

We use the notation $R_{22} \rightarrow R_{13}$ to indicate this relationship. In order to make the global state consistent, the *orphan message* m, which has been received but not sent, must be removed. This forces $P_1 \curvearrowright R_{12}$.

The situation is worst if P_3 fails:

$$P_3 \Downarrow \Rightarrow P_3 \curvearrowright R_{32} \Rightarrow P_2 \curvearrowright R_{21} \Rightarrow P_1 \curvearrowright R_{11}, P_3 \curvearrowright R_{31},$$

and the system has rolled back to its in initial state. The reason behind the domino rollbacks is the uncoordinated (independent) nature of the checkpoints taken by the different processes.

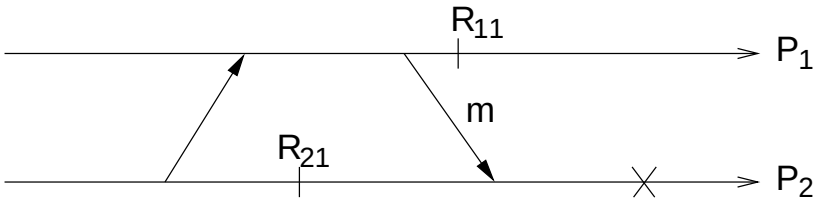
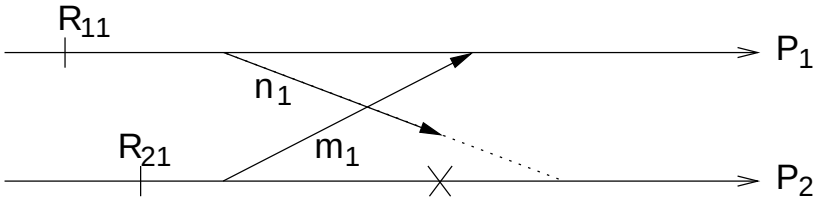


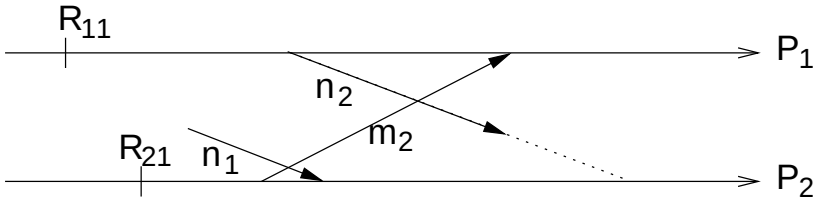
Figure 7: Rollback leading to message loss after failure of P_2

Message loss Domino rollbacks are triggered by *messages sent* by the failing process since the checkpoint to which it is rolling back. However, *messages received* since the potential recovery point also cause problems, as indicated in Figure 7. Here, $P_2 \Downarrow \Rightarrow P_2 \curvearrowright R_{21}$, leading to a consistent global state. However, the behaviour will still be incorrect, as P_1 has sent message m, while P_2 has not received m, and will never receive it. The message is lost.

However, this sort of message loss results not only from process failure but also from communication failure, a frequent event in networks. For the participating processes the two types of message loss are indistinguishable, and can be addressed by the same mechanisms: loss-resilient communication protocols.



(a) State before failure of P_2



(b) State after failure of P_2

Figure 8: State before and after failure of P_2. The message in transit (n_1) leads to livelock.

Livelock Finite communication latencies can also lead to problems, as shown in Figure 8. Failure of P_2 between sending m_1 and receiving n_1 (already sent by P_1) leads to $P_2 \curvearrowright R_{21}$, message m_1 forces $P_1 \curvearrowright R_{11}$. The latter rollback orphans message n_1, which forces $P_2 \curvearrowright R_{21}$ once more. Since P_2 managed to send m_2 before receipt of n_1, this latter rollback orphans m_2 and forces $P_1 \curvearrowright R_{11}$. With the right timing this can happen indefinitely.

Consistent Checkpointing and Recovery

The source of the domino rollback and livelock problems was that the local checkpoints were taken in an independent (uncoordinated) fashion, facing the system at recovery time with the task of finding a set of local checkpoints that together represent a *consistent cut*

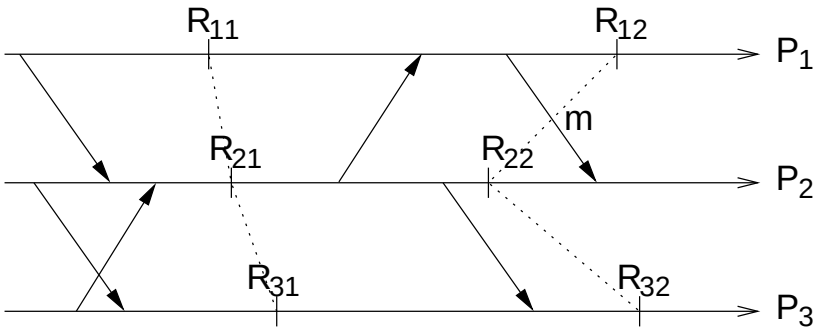


Figure 9: Strongly consistent checkpoint ($\{R_{11}, R_{21}, R_{31}\}$) and consistent checkpoint ($\{R_{12}, R_{22}, R_{32}\}$)

There are two basic kinds of such checkpoints. A *strongly consistent checkpoint*, such as the cut $\{R_{11}, R_{21}, R_{31}\}$ in Figure 9, has no messages in transit during the checkpointing interval. This requires a quiescent system during the checkpoint, and thus blocks the distributed computation for the full duration of the checkpoint.

The alternative to strongly consistent checkpoints are checkpoints representing consistent cuts, simply called *consistent checkpoints*. An example are Chandy & Lamport's *snapshots*[CL85]. Remember that this algorithm buffers messages in transit to cope with the message loss problem mentioned above. It also assumes reliable communication, which is unrealistic in most distributed system. It is preferable to use an approach that can tolerate message loss.

One simple approach is to have each node perform a local checkpoint immediately *after* sending any message to another node. While this is an independent checkpointing scheme, it is easy to see that the last local checkpoints together form a consistent checkpoint. This scheme obviously suffers from high overhead (frequent checkpointing). Any less frequent checkpointing requires system-wide coordination (synchronous checkpoints) to be able to guarantee a consistent checkpoint. (For example, checkpointing after every second message will, in general, not lead to consistent checkpoints.)

A simple synchronous checkpointing scheme providing strongly consistent checkpoints is described below. The scheme assumes ordered communication (FIFO channels), a strongly connected network (no partitioning) and some mechanism for dealing with message loss (which could be a protocol such as sliding window or nodes buffering all outgoing messages on stable storage). Each node maintains two kinds of checkpoints: a *permanent* checkpoint is part of a global checkpoint, while a *tentative* checkpoint is a candidate for being part of a global checkpoint. The checkpoint is initiated by one particular node, called the *coordinator*. Not surprisingly, the algorithm is based on the *two-phase commit* protocol[LS76] . The algorithm works as follows:

- First phase:
  1. the coordinator P_i takes a tentative checkpoint;
  2. P_i sends a t message to all other processes P_j to take tentative checkpoints;
  3. each process P_j informs P_i whether it succeeded in taking a tentative checkpoint;
  4. if P_i receives a *true* reply from each P_j it decides to make the checkpoint permanent if P_i receives at least one *false* reply it decides to discard the tentative checkpoints.
- Second phase:

1. the coordinator P_i sends a p (permanent) or u (undo) message to all other processes P_j;
2. each P_j converts or discards its tentative checkpoint accordingly;
3. a reply from P_j back to P_i can be used to let P_i know when a successful checkpoint is complete.

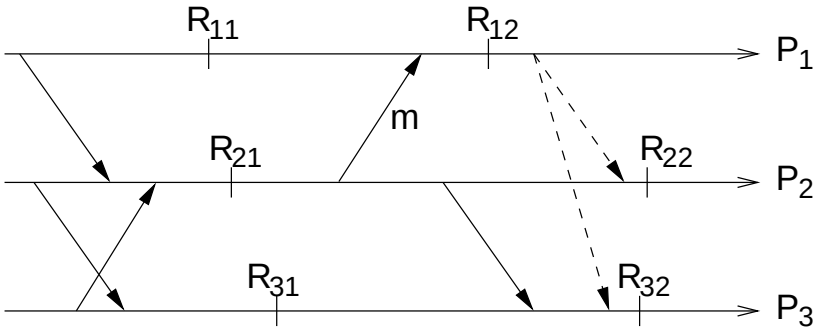


Figure 10: Synchronous checkpointing creates redundant checkpoints (R_{32})

In order to ensure consistency, this algorithm requires that processes do not send any messages other than the *true* or *false* reply between receiving the coordinator's control messages (t and p or u). This is a weak form of blocking that limits the performance of this checkpointing scheme. Furthermore, as the algorithm generates strongly consistent checkpoints, it performs redundant checkpoints if only simple consistency is required, as shown in Figure 10. Here P_1 initiates two global checkpoints, $\{R_{11}, R_{21}, R_{31}\}$ and $\{R_{12}, R_{22}, R_{32}\}$. As the cut $\{R_{12}, R_{22}, R_{31}\}$ is also consistent (but not strongly consistent), local checkpoint R_{32} is redundant. Note that a redundant local checkpoint is not necessarily bad, as it would reduce the amount of lost work in the case of a rollback.

Redundant checkpoints can be avoided by keeping track of messages sent[KT87]. In this approach, each message m is tagged with a *label* $m.l$, which is incremented for each message. Each process maintains three arrays:

- $\mathit{last\_rec}_i[j] := m.l$, where m is the *last* message received by P_i from P_j since last checkpoint ($\mathit{last\_rec}_i[j] = 0$ if no messages were received from P_j since the last checkpoint).
- $\mathit{first\_sent}_i[j] := m.l$, where m is the *first* message sent by P_i to P_j since last the checkpoint ($\mathit{first\_sent}_i[j] = 0$ if no message was sent to P_j since the last checkpoint).
- $\mathit{cohort}_i := \{j | \mathit{last\_rec}_i[j] > 0\}$ is the set of processes from which P_i has received a message (has been causally affected) since the last checkpoint.

Each process initialises *first_sent* to zero, and also maintains a variable OK which is initialised to *true*. The idea of the algorithm is that a process only needs to take a local checkpoint if the new checkpoint's coordinator has been causally affected by the process since the last permanent checkpoint. This is the case if, when receiving the t message from P_i, P_j finds that

$$\mathit{last\_rec}_i[j] \geq \mathit{first\_sent}_j[i] > 0.$$

The algorithm is as follows:

- The coordinator P_i initiates a checkpoint by:
  1. send$(t, i, \mathit{last_rec}_i[j])$ to all $P_j \in \mathit{cohort}_i$,
  2. **if** all replies are *true*, send(p) to all $P_j \in \mathit{cohort}_i$,
     else send(u) to all $P_j \in \mathit{cohort}_i$.

- Other processes, P_j, upon receiving $(t, i, last\_rec_i[j])$ do:
  1. **if** OK_j **and** $last\_rec_i[j] \geq first\_sent_j[i] > 0$
  2. take tentative checkpoint,
  3. send$(t, j, last_rec_j[k])$ to all $P_k \in cohort_j$,
  4. **if** all replies are *true*, $OK := true$ **else** $OK := false$,
  5. send(OK, j) to i.
- Upon receiving the commit message $x \in \{p, u\}$ from P_i, the other processes, P_j, do:
  1. **if** $x = p$ make permanent **else** discard tentative checkpoint,
  2. send(x, j) to all $P_k \in cohort_j$.

Note that this algorithm is expensive as it requires $O(n^2)$ messages. Recovery is initiated by the coordinator sending a *rollback* message r to all other processes. A two phase commit is used to ensure that the computation does not continue until all processes have rolled back. This leads to unnecessary rollbacks. This can again be avoided with a more sophisticated protocol which checks whether processes were causally affected.

Some more-or-less obvious improvements can be applied to consistent checkpointing. For example, explicit control messages can be avoided by tagging a sufficient amount of state information on all "normal" messages. A process will then take a tentative checkpoint (and inform its cohort) when it finds that its checkpointing state is inconsistent with that recorded in the message. This is the same idea as that of *logical time*[Mat93]. Whether this approach is beneficial is not clear *a priori*, as the reduced number of messages comes at the expense of larger messages.

Asynchronous Checkpointing and Recovery

Synchronous checkpointing as described above produces consistent checkpoints by construction, making recovery easy. In this sense it is a *pessimistic* scheme, optimised toward recovery overheads. Its drawbacks are essentially blocking the computation for the duration of a checkpoint, and the $O(n^2)$ message overhead which severely limits scalability. These features make the scheme unattractive in a scenario where failures are rare.

The alternative is to use an *optimistic* scheme, which assumes infrequent failures and consequently minimises checkpointing overheads at the expense of increased recovery overheads. In contrast to the pessimistic scheme, this approach makes no attempt at generating consistent checkpoints at normal run time, but leaves it to the recovery phase to construct a consistent state from the available checkpoints.

In this approach, checkpoints are taken locally in an independent (unsynchronised) fashion, with precautions that allow the construction of a consistent state later. Remember that orphan messages are the source of inconsistencies. These are messages that have been received but not yet sent (by the rolled-back process). The negative effect of orphaned messages can be avoided by realising that the process restarting from its last checkpoint will generate the same messages again. If the process knows that it is recovering, i.e., it is in its *roll-forward* phase (prior to reaching the previous point of failure), it can avoid the inconsistencies caused by orphan messages by suppressing any messages that it would normally have sent. Once the roll-forward proceeds past the point where the last message was sent (and received) prior to failure, the computation reverts to its normal operation. Except for timing, this results in an operation that is indistinguishable from a restart *after* the last message send operation, *provided there is no message loss*. Note that this implies a relaxation of the definition of a consistent state (but the difference is unobservable except for timing).

The requirement of suppressing outgoing messages during the roll-forward phase implies the knowledge of the number of messages sent since the last checkpoint. Hence it is necessary to *log the send count* on stable storage. Furthermore, the roll-forward cannot proceed past the receipt of any lost message, hence message loss must be avoided. This can be achieved by *logging all*

incoming messages on stable storage. During roll-forward these messages can then be replayed from the log.

This scheme works under the assumption of a *deterministic behaviour* of all processes. Such a determinism may be broken by such factors as dependence on the resident set size (the available memory may be different for different executions of the same code, leading to different caching behaviour), changed process identifiers (a restarted process may be given a different ID by the OS), or multi-threaded processes, which exhibit some inherent non-determinism. Furthermore, interrupts (resulting from I/O to local backing store) are inherently asynchronous and thus non-deterministic. One way around this problem is to checkpoint prior to handling any interrupt (or asynchronous signal)[BBG+89], although this could easily lead to excessive checkpointing. All these factors are related to the interaction with the local OS (and OS features) and can be resolved by careful implementation and appropriate OS support.

A more serious problem is that any message missing from the *replay log* will terminate the roll-back prematurely and may result in an inconsistent global state. This can be avoided by *synchronous logging* of messages, however, this slows down the computation, reducing (or eliminating) any advantage over synchronous checkpointing.

A way out of this dilemma is to be even more optimistic: using *asynchronous* (or *optimistic*) *logging*[SY85, JZ90]. Here, incoming messages are logged to volatile memory and flushed to stable memory asynchronously. On failure, any unflushed part of the log is lost, resulting in an inconsistent state. This is repaired by rolling back the orphan process(es) to a consistent state. Obviously, this can result in a domino-rollback, but the likelihood of that is significantly reduced by the asynchronous flush. Worst-case this approach is not better than independent checkpointing, but in average it is much better. One advantage is that there is no synchronisation requirement between checkpointing and logging, which greatly simplifies implementation and improves efficiency.

An optimistic checkpointing algorithm has been presented by Juang & Venkatesan[JV91]. It assumes reliable FIFO communication channels with infinite buffering capacity and finite communication delays. It considers a computation as event driven, where an *event* is defined as the *receipt of a message*. Each process is considered to be waiting for a message between events. When a message m arrives, as part of the event m is processed, an *event counter*, e, is incremented, and optionally a number of messages are sent to some directly connected nodes. Each event is logged (initially to volatile storage) as a triplet $E = \{e, m, msgs\_sent\}$, where *msgs_sent* is the set of messages sent during the event. The log is asynchronously flushed to stable storage.

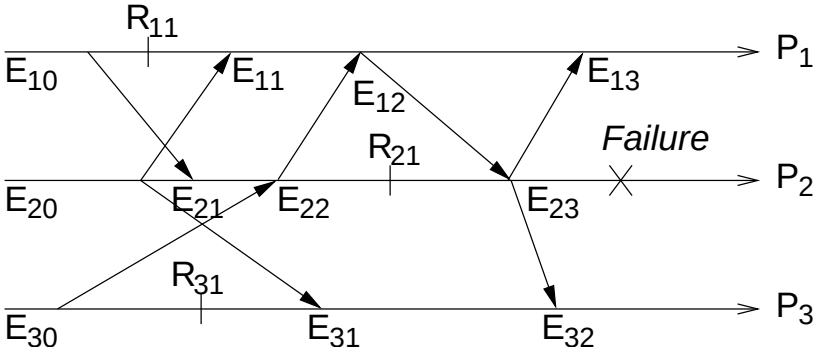


Figure 11: Optimistic checkpointing example

Figure 11 shows an example of how this works. After failure of P_2, that process can roll back to local checkpoint R_{21} and then roll forward to event E_{23}, provided that this event was logged in stable storage. If it was not logged, then E_{13} is an orphan state and a rollback to the consistent state $\{E_{12}, E_{22}, E_{31}\}$ is required. Note that E_{22} is contained in checkpoint R_{21} and thus guaranteed to be logged.

The challenge here is the detection of the latest consistent state. This is done by keeping track of the messages sent and received. Specifically, each process P_i maintains two arrays of counters:

$n\_rcvd_{i \leftarrow j}(E)$**:** the number of messages received from P_j (up to event E)

$n\_sent_{i \rightarrow j}(E)$**:** the number of messages sent to P_j (up to event E).

At restart, these are used to compare the local message count with that of the process's neighbours. If for any neighbour P_j it is found that $n\_rcvd_{j \leftarrow i} > n\_sent_{i \rightarrow j}$, then P_j is orphaned and must be rolled back until $n\_rcvd_{j \leftarrow i} \leq n\_sent_{i \rightarrow j}$. This may, of course, cause domino rollbacks of other processes.

The recovery is initiated by a restarting process, P_i, sending a broadcast message announcing its failure. P_i determines the last event, E_i, which has been recorded in its stable event log. For each process P_j receiving the failure message, E_j denotes the last local event (prior to the failure message). Each of the N processes then performs the following algorithm:

1. **for** $k := 1$ **to** N **do**
2. **for** each neighbour j **do**
3. send$(r, i, n_sent_{i \rightarrow j}(E_i))$;
4. **wait** for r messages from all neighbours;
5. **for** each message (r, j, s) received **do**
6. **if** $n\_rcvd_{i \leftarrow j}(E_i) > s$ **then** /* *have orphan* */
7. $E_i :=$ *latest* E *such that* $n\_rcvd_{i \leftarrow j}(E) = s$;

In the example of Figure 11, the following steps are taken when P_2 fails and finds that E_{22} is the last logged event (implicitly logged by R_{21}):

1. P_2 : *recover from* R_{21}; $E_2 := E_{22}$; send$(r, P_2, 2) \rightarrow P_1$; send$(r, P_2, 1) \rightarrow P_3$;
2. $P_1 \leftarrow P_2$; $E_1 := E_{13}$; $n\_rcvd_{1 \leftarrow 2}(E_{13}) = 3 > 2$: $E_1 := E_{12}$; send$(r, P_1, 2) \rightarrow P_2$;
3. $P_3 \leftarrow P_2; E_3 := E_{32}$; $n\_rcvd_{3 \leftarrow 2}(E_{32}) = 2 > 1$: $E_3 := E_{31}$; send$(r, P_3, 1) \rightarrow P_2$;
4. $P_2 \leftarrow P_1$; $n\_rcvd_{2 \leftarrow 1}(E_{22}) = 1 \leq 2$; *no change*; send$(r, P_2, 2) \rightarrow P_1$;
5. $P_2 \leftarrow P_3$; $n\_rcvd_{2 \leftarrow 3}(E_{22}) = 1 \leq 1$; *no change*; send$(r, P_2, 1) \rightarrow P_3$;

Here, $P_1 \leftarrow P_2$ is a shorthand for P_1 receiving the previously sent message from P_2. The algorithm determines the recovery state to be $\{E_{12}, E_{22}, E_{31}\}$, which is reached by rolling back to $\{R_{11}, R_{21}, R_{31}\}$ and then rolling forward by replaying the logs.

References

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[BBG+89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7:1–24, 1989.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, 1985.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[JV91] T. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 454–461. IEEE, May 1991.

[JZ90] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13:23–31, January 1987.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, 2001.

[LS76] Butler Lampson and H. Sturgis. Crash recovery in a distributed system. Working paper, Xerox PARC, Ca, USA, Ca, USA, 1976.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[PBKL95] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the 1995 USENIX Technical Conference*, pages 213–223, January 1995.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[Ske81] D. Skeen. Nonblocking commit protocols. In *SIGMOD International Conference on Management of Data*, 1981.

[SY85] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.