Cloud Computing: Theory and Practice

Dan C. Marinescu
Computer Science Division
Department of Electrical Engineering & Computer Science
University of Central Florida, Orlando, FL 32816, USA
Email: dcm@cs.ucf.edu

November 10, 2012

2 Parallel and Distributed Systems

Cloud computing is based on a large number of ideas and the experience accumulated since the first electronic computer was used to solve computationally challenging problems. In this chapter we overview concepts in parallel and distributed systems important for understanding the basic challenges in the design and use of computer clouds.

Cloud computing is intimately tied to parallel and distributed computing. Cloud applications are based on the client-server paradigm with a relatively simple software, a *thin-client*, running on the user's machine, while the computations are carried out on the cloud. Many cloud applications are data-intensive and use a number of instances which run concurrently. Transaction processing systems, e.g., web-based services, represent a large class of applications hosted by computing clouds; such applications run multiple instances of the service and require reliable and an in-order delivery of messages.

The concepts introduced in this section are very important in practice. Communication protocols which support coordination of distributed processes travel through noisy and unreliable communication channels which may lose messages or deliver duplicate, distorted, or out of order messages. To ensure reliable and in order delivery of messages the protocols stamp each message with a sequence number; in turn, a receiver sends an acknowledgment with its own sequence number to confirm the receipt of a message. The clocks of a sender and a receiver may not be synchronized thus these sequence numbers act as logical clocks. Timeouts are used to request the retransmission of lost or delayed messages.

The concept of consistent cuts and distributed snapshots are at the heart of *checkpoint-restart* procedures for long-lasting computations. Indeed, many cloud computations are data-intensive and run for extended periods of time on multiple computers in the cloud. Checkpoints are taken periodically in anticipation of the need to restart a software process when one or more systems fail; when a failure occurs the computation is restarted from the last checkpoint rather than from the beginning.

Many functions of a computer cloud require information provided by *monitors*, system components which collect state information from the individual systems. For example, controllers for cloud resource management discussed in Chapter 6 require accurate state information; security and reliability can only be implemented using information provided by specialized monitors. Coordination of multiple instances is a critical function of an application controller.

2.1 Parallel computing

As demonstrated by nature, the ability to work in parallel as a group represents a very efficient way to reach a common target; human beings have learned to aggregate themselves, and to assemble man-made devices in organizations where each entity may have modest ability, but a network of entities can organize themselves to accomplish goals that an individual entity cannot. Thus, we should not be surprised that the thought that individual systems should work in concert for solving complex applications was formulated early on in the computer age.

Parallel computing allows us to solve large problems by splitting them into smaller ones and solving them concurrently. Parallel computing was considered for many years the holy

grail for solving data-intensive problems encountered in many areas of science, engineering, and enterprise computing; it required major advances in several areas including, algorithms, programming languages and environments, performance monitoring, computer architecture, interconnection networks, and, last but not least, solid-state technologies.

Parallel hardware and software systems allow us to solve problems demanding more resources than those provided by a single system and, at the same time, to reduce the time required to obtain a solution. The speed-up measures the effectiveness of parallelization; in the general case the *speed-up* of the parallel computation is defined as

$$S(N) = \frac{T(1)}{T(N)}, \tag{1}$$

with $T(1)$ the execution time of the sequential computation and $T(N)$ the execution time when N parallel computations are carried out. Amdahl's Law[7] gives the potential speed-up of a parallel computation; it states that the portion of the computation which cannot be parallelized determines the overall speed-up. If α is the fraction of running time a sequential program spends on non-parallelizable segments of the computation then

$$S = \frac{1}{\alpha}. \tag{2}$$

To prove this result call σ the sequential time and π the parallel time and start from the definitions of $T(1), T(N)$, and α:

$$T(1) = \sigma + \pi, \qquad T(N) = \sigma + \frac{\pi}{N}, \qquad \text{and} \qquad \alpha = \frac{\sigma}{\sigma + \pi}. \tag{3}$$

Then

$$S = \frac{T(1)}{T(N)} = \frac{\sigma + \pi}{\sigma + \pi/N} = \frac{1 + \pi/\sigma}{1 + (\pi/\sigma) \times (\ 1/N)}. \tag{4}$$

But

$$\pi/\sigma = \frac{1 - \alpha}{\alpha} \tag{5}$$

Thus, for large N

$$S = \frac{1 + (1 - \alpha)/\alpha}{1 + (1 - \alpha)/(N\alpha)} = \frac{1}{\alpha + (1 - \alpha)/N} \approx \frac{1}{\alpha} \tag{6}$$

Amdahl's law applies to a *fixed problem size*; in this case the amount of work assigned to each one of the parallel processes decreases when the number of processes increases and this affects the efficiency of the parallel execution.

When the problem size is allowed to change Gustafson's Law gives the *scaled speed-up* with N parallel processes as

$$S(N) = N - \alpha(N - 1). \tag{7}$$

[7]Gene Amdahl is a theoretical physicist turned computer architect who contributed significantly to the development of several IBM systems including System/360 and then started his own company, Amdahl Corporation; his company produced high performance systems in the 1970s. Amdahl is best known for Amdahl's Law formulated in 1960.

As before, we call σ the sequential time; now π is the *fixed parallel time per process*; α is given by Equation 3. The sequential execution time, $T(1)$, and the parallel execution time with N parallel processes, $T(N)$, are

$$T(1) = \sigma + N\pi \quad \text{and} \quad T(N) = \sigma + \pi. \tag{8}$$

Then the scaled speed-up is

$$S(N) = \frac{T(1)}{T(N)} = \frac{\sigma + N\pi}{\sigma + \pi} = \frac{\sigma}{\sigma + \pi} + \frac{N\pi}{\sigma + \pi} = \alpha + N(1 - \alpha) = N - \alpha(N - 1). \tag{9}$$

Amdahl's Law expressed by Equation 2 and the *scaled speed-up* given by Equation 7 assume that all processes are assigned the same amount of work. The scaled speed-up assumes that the amount of work assigned to each process is the same regardless of the problem size. Then to maintain the same execution time the number of parallel processes must increase with the problem size. The scaled speed-up captures the essence of efficiency, namely that the limitations of the sequential part of a code can be balanced by increasing the problem size.

Coordination of concurrent computations could be quite challenging and involves overhead which ultimately reduces the speed-up of parallel computations. Often the parallel computation involves multiple stages and all concurrent activities must finish one stage before starting the execution of the next one; this *barrier synchronization* further reduce the speed-up.

The subtasks of a parallel program are called *processes*, while *threads* are light-weight subtasks. Concurrent execution could be very challenging, e.g., it could lead to *race conditions*, an undesirable effect when the results of concurrent execution depend on the sequence of events. Often, shared resources must be protected by *locks* to ensure serial access. Another potential problem for concurrent execution of multiple processes/threads is the presence of deadlocks; a *deadlock* occurs when processes/threads competing with one another for resources are forced to wait for additional resources held by other processes/threads and none of the processes/threads can finish. The four Coffman conditions, must hold simultaneously for a deadlock to occur:

1. *Mutual exclusion;* at least one resource must be non-sharable, only one process/thread may use the resource at any given time.

2. *Hold and wait;* at least one processes/thread must hold one or more resources and wait for others.

3. *No-preemption;* the scheduler or a monitor should not be able to force a process/thread holding a resource to relinquish it.

4. *Circular wait;* given the set of n processes/threads $\{P_1, P_2, P_3, \ldots, P_n\}$, P_1 should wait for a resource held by P_2, P_2 should wait for a resource held by P_3, and so on, P_n should wait for a resource held by P_1.

There are others potential problems related to concurrency. When two or more processes/threads continually change their state in response to changes in the other processes

we have a *livelock* condition; the result is that none of the processes can complete its execution. Very often processes/threads running concurrently are assigned priorities and scheduled based on these priorities. *Priority inversion* occurs when a higher priority process/task is indirectly preempted by a lower priority one.

Concurrent processes/task can communicate using messages or shared memory. Multicore processors sometimes use shared memory but the shared-memory is seldom used in modern supercomputers because shared-memory systems are not scalable. Message passing is the communication method used exclusively in large-scale distributed systems and our discussion is restricted to this communication paradigm.

Shared memory is extensively used by the system software; the stack is an example of shared memory used to save the state of a process or thread. The kernel of an operating system uses control structures such as processor and core tables for multiprocessor and multicore system management, process and thread tables for process/thread management, page tables for virtual memory management, and so on. Multiple application threads running on a multi-core processor often communicate via the shared memory of the system. Debugging a a message passing application is considerably easier than debugging a shared memory one.

We distinguish the *fine-grain* from the *coarse-grain* parallelism; in the former case relatively small blocks of the code can be executed in parallel without the need to communicate or synchronize with other threads or processes, while in the latter case large blocks of code can be executed in parallel. The speed-up of applications displaying fine-grain parallelism is considerably lower that those of coarse-grained applications; indeed, the processor speed is orders of magnitude higher than the communication speed even on systems with a fast interconnect.

In many cases discovering parallelism is quite challenging and the development of parallel algorithms requires a considerable effort. For example, many numerical analysis problems such as solving large systems of linear equations, or solving systems of PDEs (Partial Differential Equations) require algorithms based on domain decomposition methods.

Data parallelism is based on partitioning the data into several blocks and running multiple copies of the same program concurrently, each running on a different data block, thus the name of the paradigm, Same Program Multiple Data (SPMD).

Decomposition of a large problem into a set of smaller problems that can be solved concurrently is sometimes trivial. For example, assume that we wish to manipulate the display of a three-dimensional object represented as a *3D* lattice of $(n \times n \times n)$ points; to rotate the image we would apply the same transformation to each one of the n^3 points. Such a transformation can be done by a geometric engine, a hardware component which can carry out the transformation of a subset of n^3 points concurrently.

Suppose that we want to search for the occurrence of an object in a set of n images, or of a string of characters in n records; such a search can be conducted in parallel. In all these instances the time required to carry out the computational task using N processing elements is reduced by a factor of N.

A very appealing class of applications of cloud computing are numerical simulations of complex systems which require an optimal design; in such instances multiple design alternatives must be compared and optimal ones selected based on several optimization criteria. Consider for example the design of a circuit using FPGAs. An FPGA (Field Programmable Gate Array) is an integrated circuit designed to be configured by the customer using a hardware description language (HDL), similar to that used for an application-specific integrated

circuit (ASIC). As multiple choices for the placement of components and for interconnecting them exist, the designer could run concurrently N versions of the design choices and choose the one with the best performance, e.g., minimum power consumption. Alternative optimization objectives could be to reduce cross-talk among the wires or to minimize the overall noise. Each alternative configuration requires hours, or maybe days of computing hence, running them concurrently reduces the design time considerably.

The list of companies which aimed to support parallel computing and ended up as a casualty of this effort is long and includes names such as: Ardent, Convex, Encore, Floating Point Systems, Inmos, Kendall Square Research, MasPar, nCube, Sequent, Tandem, and Thinking Machines. The difficulties to develop new programming models and the effort to design programming environments for parallel applications added to the challenges faced by all these companies.

From the very beginning it was clear that parallel computing requires specialized hardware and system software. It was also clear that the interconnection fabric was critical for the performance of parallel computing systems. We now take a closer look at the parallelism at different levels and the means to exploit it.

2.2 Parallel computer architecture

Our discussion of parallel computer architectures starts with the recognition that parallelism at different levels can be exploited; these levels are:

1. *Bit level parallelism.* The number of bits processed per clock cycle, often called a word size, has increased gradually from 4-bit processors to 8-bit, 16-bit, 32-bit, and since 2004 to 64-bit. This has reduced the number of instructions required to process larger size operands and allowed a significant performance improvement. During this evolutionary process the number of address bits have also increased allowing instructions to reference a larger address space.

2. *Instruction-level parallelism.* Today's computers use multi-stage processing pipelines to speed up execution. Once an n-stage pipeline is full, an instruction is completed at every clock cycle. A "classic" pipeline of a RISC (Reduced Instruction Set Computing) architecture consists of five stages[8]: instruction fetch, instruction decode, instruction execution, memory access, and write back. A CISC (Complex Instruction Set Computing) architecture could have a much large number of pipelines stages, e.g., an Intel Pentium 4 processor has a 35-stage pipeline.

3. *Data parallelism or loop parallelism.* The program loops can be processed in parallel.

4. *Task parallelism.* The problem can be decomposed into tasks that can be carried out concurrently. ~~A widely used type of task parallelism is the SPMD (Same Program Multiple Data) paradigm. As the name suggests individual processors run the same program but on different segments of the input-data. Data dependencies cause different flows of control in individual tasks.~~

[8]The number of pipeline stages in different RISC processors varies. For example, ARM7 and earlier implementations of ARM processors have a three stage pipeline, fetch, decode, and execute. Higher performance designs, such as the ARM9, have deeper pipelines: Cortex-A8 has thirteen stages.

In 1966 Michael Flynn proposed a classification of computer architectures based on the number of *concurrent control/instruction* and *data streams*: SISD (Single Instruction Single Data), SIMD (Single Instruction, Multiple Data), and MIMD (Multiple Instructions, Multiple Data)[9].

The SIMD architecture supports vector processing. When a SIMD instruction is issued, the operations on individual vector components are carried out concurrently. For example, to add two vectors $(a_1, a_2 \ldots\ldots a_{50})$ and $(b_1, b_2 \ldots\ldots b_{50})$, all 50 pairs of vector elements are added concurrently and all the sums $(a_i + b_i), 1 \leq i \leq 50$ are available at the same time.

The first use of SIMD instructions was in vector supercomputers such as the CDC Star-100 and the Texas Instruments ASC in the early 1970s. Vector processing was especially popularized by Cray in the 1970s and 1980s, by attached vector processors such as those produced by the FPS (Floating Point Systems), and by supercomputers such as the Thinking Machines CM-1 and CM-2. Sun Microsystems introduced SIMD integer instructions in its "VIS" instruction set extensions in 1995, in its UltraSPARC I microprocessor; the first widely-deployed SIMD for gaming was Intel's MMX extensions to the *x86* architecture. IBM and Motorola then added AltiVec to the POWER architecture and there have been several extensions to the SIMD instruction sets for both architectures.

The desire to support real-time graphics with vectors of two, three, or four dimensions led to the development of Graphic Processing Units (GPU). GPUs are very efficient at manipulating computer graphics, and their highly parallel structures based on SIMD execution support parallel processing of large blocks of data. GPUs produced by Intel, NVIDIA, and AMD/ATI are used in embedded systems, mobile phones, personal computers, workstations, and game consoles.

A MIMD architecture refers to a system with several processors that function asynchronously and independently; at any time, different processors may be executing different instructions on different data. The processors can share a common memory of a MIMD and we distinguish several types of systems, Uniform Memory Access (UMA), Cache Only Memory Access (COMA), and Non-Uniform Memory Access (NUMA). A MIMD system could have a distributed memory; in this case the processors and the memory communicate with one another using an interconnection network, such as a hypercube, a `2D` torus, a `3D` torus, an omega network, or another network topology. Today, most supercomputers are MIMD machines and some use GPUs instead of traditional processors. Multi-core processors with multiple processing units are now ubiquitous.

Modern supercomputers derive their power from architecture and parallelism rather than the increase of processor speed. The supercomputers of today consist of very large number of processors and cores communicating via very fast custom interconnects. In mid 2012 the most powerful supercomputer was a Linux-based IBM Sequoia-BlueGene/Q system powered by Power BQC 16-core processors running at 1.6 GHz. The system, installed at Lawrence Livermore National Laboratory, has a total of 1 572 864 cores and 1 572 864 GB of memory, archives a sustainable speed of 16.32 petaflops, and consumes 7.89 MW of power. Several most powerful systems listed in the "Top 500 supercomputers" (see *http://www.top500.org/*) are powered by NVIDIA 2050 GPU; three of the top 10 use an InfiniBand interconnect.

The next natural step was triggered by advances in communication networks when low-latency and high-bandwidth WANs (Wide Area Networks) allowed individual systems, many

[9]Another category, MISD (Multiple Instructions Single Data) is a fourth possible architecture, but it is very rarely used, mostly for fault tolerance.

of them multiprocessors, to be geographically separated. Large-scale distributed systems were first used for scientific and engineering applications and took advantage of the advancements in system software, programming models, tools, and algorithms developed for parallel processing.

2.3 Distributed systems

A *distributed system* is a collection of autonomous computers, connected through a network and distribution software called *middleware*, which enables computers to coordinate their activities and to share the resources of the system; the users perceive the system as a single, integrated computing facility.

A distributed system has several characteristics: its components are autonomous, scheduling and other resource management and security policies are implemented by each system, there are multiple points of control and multiple points of failure, and the resources may not be accessible at all times. Distributed systems can be scaled by adding additional resources and can be designed to maintain availability even at low levels of hardware/software/network reliability.

Distributed systems have been around for several decades. For example, distributed file systems and network file systems have been used for user convenience and to improve reliability and functionality of file systems for many years. Modern operating systems allow a user to *mount* a remote file system and access it the same way a local file system is accessed, yet with a performance penalty due to larger communication costs. The *Remote Procedure Call* (RPC) supports inter-process communication and allows a procedure on a system to invoke a procedure running in a different address space, possibly on a remote system. RPCs were introduced in the early 1970s by Bruce Nelson and used for the first time at Xerox; the Network File System (NFS) introduced in 1984 was based on Sun's RPC. Many programming languages support RPCs; for example, Java Remote Method Invocation (Java RMI) provides a functionality similar to the one of UNIX RPC methods, XML-RPC uses XML to encode HTML-based calls.

The middleware should support a set of desirable properties of a distributed system:

- Access transparency - local and remote information objects are accessed using identical operations;
- Location transparency - information objects are accessed without knowledge of their location;
- Concurrency transparency - several processes run concurrently using shared information objects without interference among them;
- Replication transparency - multiple instances of information objects are used to increase reliability without the knowledge of users or applications;
- Failure transparency - the concealment of faults;
- Migration transparency - the information objects in the system are moved without affecting the operation performed on them;

- Performance transparency - the system can be reconfigured based on the load and quality of service requirements;
- Scaling transparency - the system and the applications can scale without a change in the system structure and without affecting the applications.

2.4 Global state of a process group

To understand the important properties of distributed systems we use a model, an abstraction based on two critical components, processes and communication channels. A *process* is a program in execution and a *thread* is a light-weight process. A thread of execution is the smallest unit of processing that can be scheduled by an operating system.

A process is characterized by its *state*; the state is the ensemble of information we need to restart a process after it was suspended. An *event* is a change of state of a process. The events affecting the state of process p_i are numbered sequentially as $e_i^1, e_i^2, e_i^3, \ldots$ as shown in the space-time diagram in Figure 4(a). A process p_i is in state σ_i^j immediately after the occurrence of event e_i^j and remains in that state until the occurrence of the next event, e_i^{j+1}.

A *process group* is a collection of cooperating processes; these processes work in concert and communicate with one another in order to reach a common goal. For example a parallel algorithm to solve a system of partial deferential equations (PDEs) over a domain D may partition the data in several segments and assign each segment to one of the members of the process group. The processes in the group must cooperate with one another and iterate until the common boundary values computed by one process agree with the common boundary values computed by another.

A *communication channel* provides the means for processes or threads to communicate with one another and coordinate their actions by exchanging messages. Without loss of generality we assume that communication among processes is done only by means of *send(m)* and *receive(m)* communication events, where m is a message. We use the term "message" for a structured unit of information, which can be interpreted only in a semantic context by the sender and the receiver. The *state of a communication channel* is defined as follows: given two processes p_i and p_j, the state of the channel, $\xi_{i,j}$, from p_i to p_j consists of messages sent by p_i but not yet received by p_j.

These two abstractions allow us to concentrate on critical properties of distributed systems without the need to discuss the detailed physical properties of the entities involved. The model presented is based on the assumption that a channel is a unidirectional bit pipe of infinite bandwidth and zero latency, but unreliable; messages sent through a channel may be lost, distorted, or the channel may fail, lose its ability to deliver messages. We also assume that the time a process needs to traverse a set of states is of no concern and that processes may fail, or be aborted.

A *protocol* is a finite set of messages exchanged among processes to help them coordinate their actions. Figure 4(c) illustrates the case when communication events are dominant in the local history of processes, p_1, p_2 and p_3. In this case only e_1^5 is a local event; all others are communication events. The particular protocol illustrated in Figure 4(c) requires processes p_2 and p_3 to send messages to the other processes in response to a message from process p_1.

The informal definition of the state of a single process can be extended to collections of communicating processes. The *global state of a distributed system* consisting of several

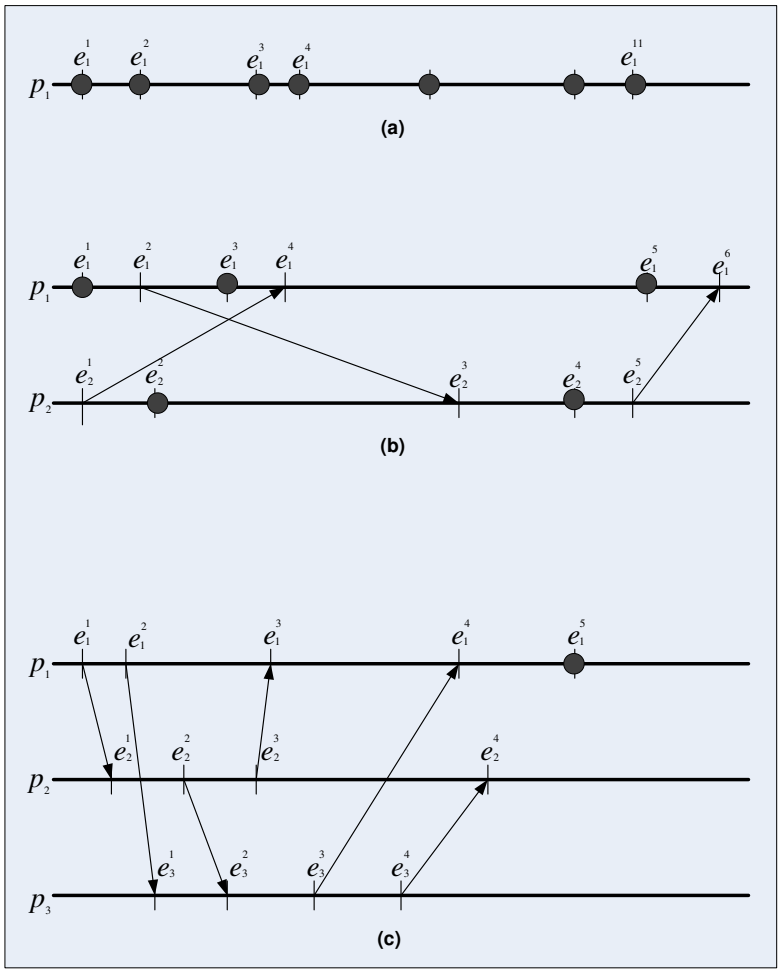

Figure 4: Space-time diagrams display local and communication events during a process lifetime. Local events are small black circles. Communication events in different processes are connected by lines originating at a *send* event and terminated by an arrow at the *receive* event. (a) All events in case of a single process p_1 are local; the process is in state σ_1 immediately after the occurrence of event e_1^1 and remains in that state until the occurrence of event e_1^2. (b) Two processes p_1 and p_2; event e_1^2 is a communication event, p_1 sends a message to p_2; event e_2^3 is a communication event, process p_2 receives the message sent by p_1. (c) Three processes interact by means of communication events.

processes and communication channels is the union of the states of the individual processes and channels [34].

Call h_i^j the history of process p_i up to and including its j-th event, e_i^j, and call σ_i^j the local state of process p_i following event e_i^j. Consider a system consisting of n processes, $p_1, p_2, \ldots, p_i, \ldots, p_n$ with $\sigma_i^{j_i}$ the local state of process p_i; then, the global state of the system is an n-tuple of local states

$$\Sigma^{(j_1,j_2,\cdots,j_n)} = (\sigma_1^{j_1}, \sigma_2^{j_2}, \ldots, \sigma_i^{j_i}, \ldots, \sigma_n^{j_n}). \tag{10}$$

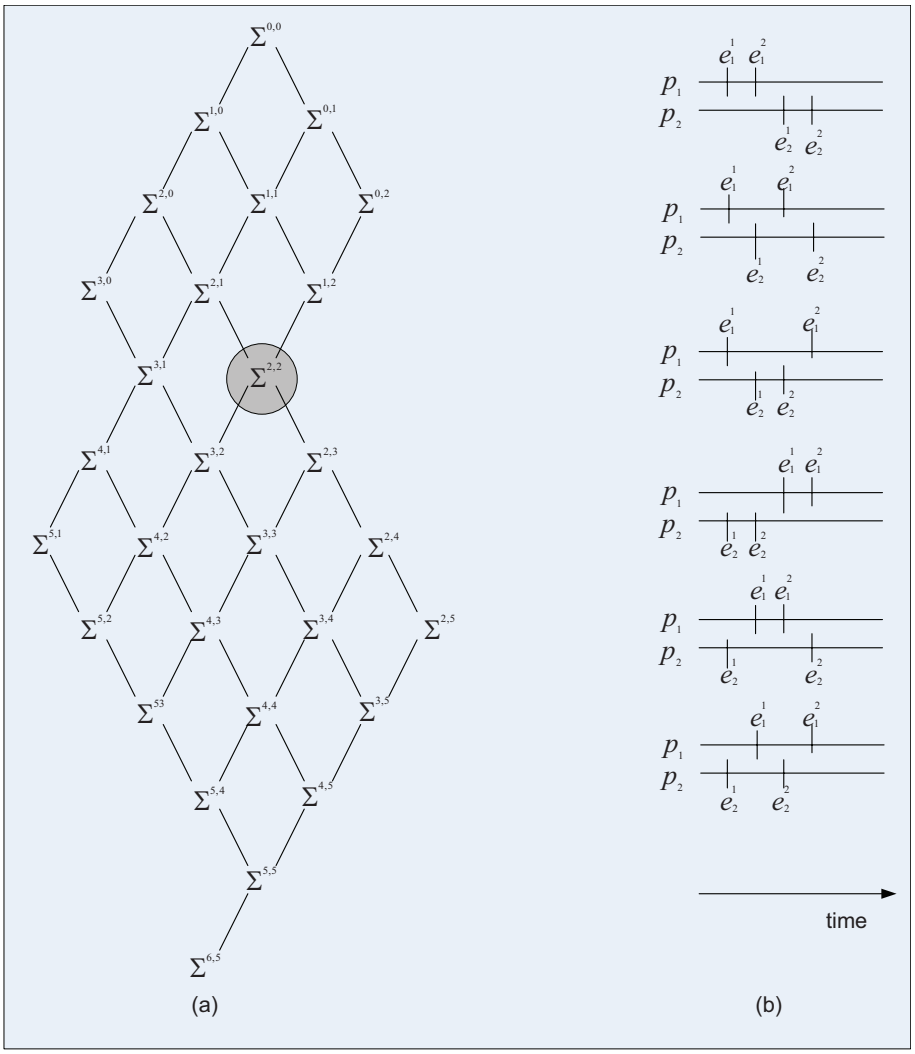

Figure 5: (a) The lattice of the global states of two processes with the space-time diagrams in Figure 4(b). (b) The six possible sequences of events leading to the state $\Sigma^{(2,2)}$.

The state of the channels does not appear explicitly in this definition of the global state because the state of the channels is encoded as part of the local state of the processes communicating through the channels.

The global states of a distributed computation with n processes form an n-dimensional lattice. The elements of this lattice are global states $\Sigma^{(j_1,j_2,\ldots,j_n)}(\sigma_1^{j_1}, \sigma_2^{j_2}, \ldots, \sigma_n^{j_n})$.

Figure 5(a) shows the lattice of global states of the distributed computation in Figure 4(b). This is a two-dimensional lattice because we have two processes, p_1 and p_2. The lattice of global states for the distributed computation in Figure 4(c) is a three-dimensional lattice, the computation consists of three concurrent processes, p_1, p_2, and p_3.

The initial state of the system in Figure 5(b) is the state before the occurrence of any event and it is denoted by $\Sigma^{(0,0)}$; the only global states reachable from $\Sigma^{(0,0)}$ are $\Sigma^{(1,0)}$, and $\Sigma^{(0,1)}$. The communication events limit the global states the system may reach; in this example the system cannot reach the state $\Sigma^{(4,0)}$ because process p_1 enters state σ_4 only

after process p_2 has entered the state σ_1. Figure 5(b) shows the six possible sequences of events to reach the global state $\Sigma^{(2,2)}$:

$$(e_1^1, e_1^2, e_2^1, e_2^2), (e_1^1, e_2^1, e_1^2, e_2^2), (e_1^1, e_2^1, e_2^2, e_1^2), (e_2^1, e_2^2, e_1^1, e_1^2), (e_2^1, e_1^1, e_1^2, e_2^2), (e_2^1, e_1^1, e_2^2, e_1^2). \quad (11)$$

An interesting question is how many paths to reach a global state exist; the more paths exist, the harder is to identify the events leading to a state when we observe an undesirable behavior of the system. A large number of paths increase the difficulties to debug the system.

We conjecture that in the case of two threads in Figure 5(a) the number of paths from the global state $\Sigma^{(0,0)}$ to $\Sigma^{(m,n)}$ is

$$N_p^{(m,n)} = \frac{(m+n)!}{m!n!}. \quad (12)$$

We have already seen that there are 6 paths leading to state $\Sigma^{(2,2)}$ and, indeed

$$N_p^{(2,2)} = \frac{(2+2)!}{2!2!} = \frac{24}{4} = 6. \quad (13)$$

To prove Equation 12 we use a method resembling induction; we notice first that the global state $\Sigma^{(1,1)}$ can only be reached from the states $\Sigma^{(1,0)}$ and $\Sigma^{(0,1)}$ and that $N_p^{(1,1)} = (2)!/1!1! = 2$ thus, the formula is true for $m = n = 1$. Then we show that if the formula is true for the $(m-1, n-1)$ case it will also be true for the (m, n) case. If our conjecture is true then

$$N_p^{[(m-1),n]} = \frac{[(m-1)+n)]!}{(m-1)!n!} \quad (14)$$

and

$$N_p^{[m,(n-1)]} = \frac{[(m+(n-1)]!}{m!(n-1)!}. \quad (15)$$

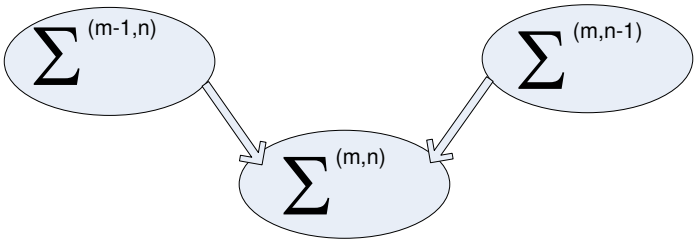

Figure 6: In the two dimensional case the global state $\Sigma^{(m,n)}$, $\forall(m,n) \geq 1$ can only be reached from two states, $\Sigma^{(m-1,n)}$ and $\Sigma^{(m,n-1)}$.

We observe that the global state $\Sigma^{(m,n)}$, $\forall(m,n) \geq 1$ can only be reached from two states, $\Sigma^{(m-1,n)}$ and $\Sigma^{(m,n-1)}$, see Figure 6 thus

$$N_p^{(m,n)} = N_p^{(m-1,n)} + N_p^{(m,n-1)}. \tag{16}$$

It is easy to see that indeed

$$\begin{aligned} \frac{[(m-1)+n]!}{(m-1)!n!} + \frac{[m+(n-1)]!}{m!(n-1)!} &= (m+n-1)! \left[\frac{1}{(m-1)!n!} + \frac{1}{m!(n-1)!}\right] \\ &= \frac{(m+n)!}{m!n!}. \end{aligned} \tag{17}$$

This shows that our conjecture is true thus, Equation 12 gives the number of paths to reach the global state $\Sigma^{(m,n)}$ from $\Sigma^{(0,0)}$ when two threads are involved. This expression can be generalized for the case of q threads; using the same strategy it is easy to see that the number of path from the state $\Sigma^{(0,0,\ldots,0)}$ to the global state $\Sigma^{(n_1,n_2,\ldots,n_q)}$ is

$$N_p^{(n_1,n_2,\ldots,n_q)} = \frac{(n_1+n_2+\ldots+n_q)!}{n_1!n_2!\ldots n_q!} \tag{18}$$

Indeed, it is easy to see that

$$N_p^{(n_1,n_2,\ldots,n_q)} = N_p^{(n_1-1,n_2,\ldots,n_q)} + N_p^{(n_1,n_2-1,\ldots,n_q)} + \ldots + N_p^{(n_1,n_2,\ldots,n_q-1)} \tag{19}$$

Equation 18 gives us an indication on how difficult it is to debug a system with a large number of concurrent threads.

Many problems in distributed systems are instances of the *global predicate evaluation problem* (GPE) where the goal is to evaluate a Boolean expression whose elements are functions of the global state of the system.

2.5 Communication protocols and process coordination

A major concern in any parallel and distributed system is communication in the presence of channel failures. There are multiple modes for a channel to fail and some lead to messages being lost. In the general case, it is impossible to guarantee that two processes will reach an agreement in case of channel failures, see Figure 7.

Given two processes p_1 and p_2 connected by a communication channel that can lose a message with probability $\epsilon > 0$, no protocol capable of guaranteeing that two processes will reach agreement exists, regardless of how small the probability ϵ is.

The proof of this statement is by contradiction; assume that such a protocol exists and it consists of n messages; recall that a protocol is a finite sequence of messages. Since any message might be lost with probability ϵ the protocol should be able to function when only $n-1$ messages reach their destination, the last one being lost. Induction on the number of messages proves that indeed no such protocol exists; indeed, the same reasoning leads us to conclude that the protocol should function correctly with $(n-2)$ messages, and so on.

In practice, error detection and error correction codes allow processes to communicate reliably though noisy digital channels. The redundancy of a message is increased by more bits and packaging a message as a codeword; the recipient of the message is then able to decide if the sequence of bits received is a valid codeword and, if the code satisfies some distance properties, then the recipient of the message is able to extract the original message from a bit string in error.

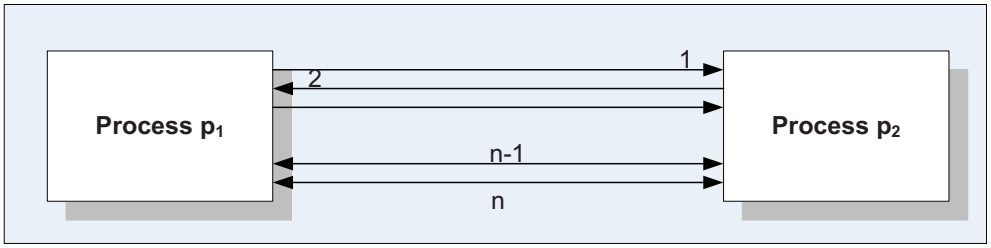

Figure 7: Process coordination in the presence of errors; each message may be lost with probability ϵ. If a protocol consisting of n messages exists, then the protocol should be able to function properly with $n-1$ messages reaching their destination, one of them being lost.

Communication protocols implement not only *error control* mechanisms, but also flow control and congestion control. *Flow control* provides feedback from the receiver, it forces the sender to transmit only the amount of data the receiver is able to buffer and then process. *Congestion control* ensures that the offered load of the network does not exceed the network capacity. In store-and-forward networks individual routers may drop packets when the network is congested and the sender is forced to retransmit. Based on the estimation of the RTT (Round-Trip-Time) the sender can detect congestion and reduce the transmission rate.

The implementation of these mechanisms require the measurement of *time intervals*, the time elapsed between two events; we also need a *global concept of time* shared by all entities that cooperate with one another. For example, a computer chip has an *internal clock* and a predefined set of actions occurs at each clock tick. Each chip has an *interval timer* that helps enhance the system's fault tolerance; when the effects of an action are not sensed after a predefined interval, the action is repeated.

When the entities communicating with each other are networked computers, the precision of the clock synchronization is critical [205]. The event rates are very high, each system goes through state changes at a very fast pace; modern processors run at a $2-4$ GHz clock rate. That explains why we need to measure time very accurately; indeed, we have atomic clocks with an accuracy of about 10^{-6} seconds per year.

An isolated system can be characterized by its *history* expressed as a sequence of events, each event corresponding to a change of the state of the system. Local timers provide relative time measurements. A more accurate description adds to the system's history the time of occurrence of each event as measured by the local timer.

Messages sent by processes may be lost or distorted during transmission. Without additional restrictions regarding message delays and errors there are no means to ensure a perfect synchronization of local clocks and there are no obvious methods to ensure a global ordering of events occurring in different processes. Determining the global state of a large-scale distributed system is a very challenging problem.

The mechanisms described above are insufficient once we approach the problem of cooperating entities. To coordinate their actions, two entities need a common perception of time. Timers are not enough, clocks provide the only way to measure distributed duration, that is, actions that start in one process and terminate in another. *Global agreement on time* is necessary to *trigger actions* that should occur concurrently, e.g., in a real-time control

system of a power plant several circuits must be switched on at the same time. Agreement on *the time when events occur* is necessary for distributed recording of events, for example, to determine a precedence relation through a temporal ordering of events. To ensure that a system functions correctly we need to determine that the event causing a change of state occurred before the state change, e.g., the sensor triggering an alarm has to change its value before the emergency procedure to handle the event was activated. Another example of the need for agreement on the time of occurrence of events is in replicated actions. In this case several replicas of a process must log the time of an event in a consistent manner.

Timestamps are often used for event ordering using a global time-base constructed on local virtual clocks [235]. The Δ-protocols [94] achieve total temporal order using a global time base. Assume that local virtual clock readings do not differ by more than π, called *precision* of the global time base. Call g the *granularity of physical clocks*. First, observe that the granularity should not be smaller than the precision; given two events a and b occurring in different processes if $t_b - t_a \leq \pi + g$ we cannot tell which of a or b occurred first [359]. Based on these observations, it follows that the order discrimination of clock-driven protocols cannot be better than twice the clock granularity.

System specification, design, and analysis require a clear understanding of *cause-effect relationships*. During the system specification phase we view the system as a state machine and define the actions that cause transitions from one state to another. During the system analysis phase we need to determine the cause that brought the system to a certain state.

The activity of any process is modeled as a sequence of *events*; hence, the binary relation cause-effect should be expressed in terms of events and should express our intuition that *the cause must precede the effects*. Again, we need to distinguish between local events and communication events. The latter ones affect more than one process and are essential for constructing a global history of an ensemble of processes. Let h_i denote the local history of process p_i and let e_i^k denote the k-th event in this history.

The binary cause-effect relationship between two events has the following properties:

1. Causality of local events can be derived from the process history:

$$\text{if } \; e_i^k, e_i^l \in h_i \; \text{ and } \; k < l \; \text{ then } \; e_i^k \rightarrow e_i^l. \tag{20}$$

2. Causality of communication events:

$$\text{if } \; e_i^k = send(m) \; \text{ and } \; e_j^l = receive(m) \; \text{ then } \; e_i^k \rightarrow e_j^l. \tag{21}$$

3. Transitivity of the causal relationship:

$$\text{if } \; e_i^k \rightarrow e_j^l \; \text{ and } \; e_j^l \rightarrow e_m^n \; \text{ then } e_i^k \rightarrow e_m^n. \tag{22}$$

Two events in the global history may be unrelated, neither one is the cause of the other; such events are said to be *concurrent events*.

2.6 Logical clocks

A *logical clock (LC)* is an abstraction necessary to ensure the clock condition in the absence of a global clock. Each process p_i maps events to positive integers. Call $LC(e)$ the local variable associated with event e. Each process time-stamps each message m sent with the value of the logical clock at the time of sending, $TS(m) = LC(send(m))$. The rules to update the logical clock are specified by the following relationship:

$$LC(e) := \begin{cases} LC + 1 & \text{if } e \text{ is a local event or a } send(m) \text{ event} \\ \max(LC, TS(m) + 1) & \text{if } e = receive(m). \\ \text{max(LC+1, TS(m)+1)} & \end{cases} \tag{23}$$

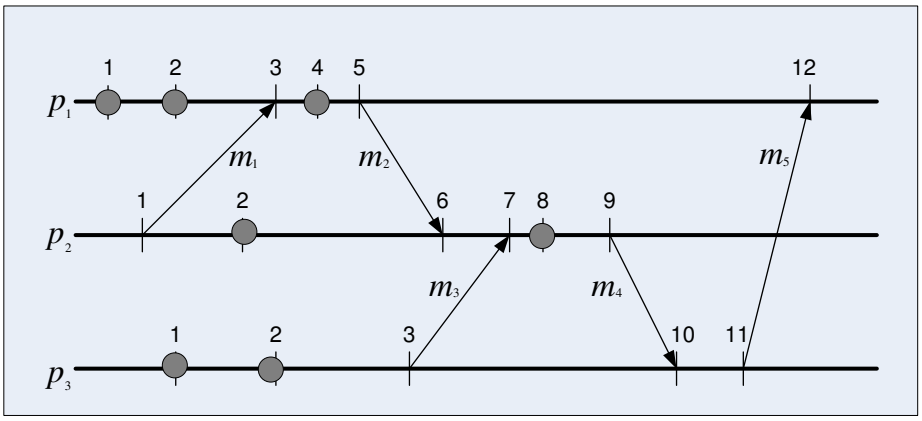

Figure 8: Three processes and their logical clocks; The usual labeling of events as $e_1^1, e_1^2, e_1^3, \ldots$ is omitted to avoid overloading the figure; only the logical clock values for the local and for the communication events are marked. The correspondence between the events and the logical clock values is obvious: $e_1^1, e_2^1, e_3^1 \rightarrow 1$, $e_1^5 \rightarrow 5$, $e_2^4 \rightarrow 7$, $e_3^4 \rightarrow 10$, $e_1^6 \rightarrow 12$, and so on. Global ordering of all events is not possible; there is no way to establish the ordering of events e_1^1, e_2^1 and e_3^1.

The concept of logical clocks is illustrated in Figure 8 using a modified *space-time diagram* where the events are labeled with the logical clock value. Messages exchanged between processes are shown as lines from the sender to the receiver; the communication events corresponding to sending and receiving messages are marked on these diagrams.

Each process labels local events and sends events sequentially until it receives a message marked with a logical clock value larger than the next local logical clock value, as shown in Equation 23. It follows that logical clocks do not allow a global ordering of all events. For example, there is no way to establish the ordering of events e_1^1, e_2^1 and e_3^1 in Figure 8. Nevertheless, communication events allow different processes to coordinate their logical clocks; for example, process p_2 labels the event e_2^3 as 6 because of message m_2, which carries the information about the logical clock value as 5 at the time message m_2 was sent. Recall that e_i^j is the j-th event in process p_i.

Logical clocks lack an important property, *gap detection*; given two events e and e' and their logical clock values, $LC(e)$ and $LC(e')$, it is impossible to establish if an event e'' exists such that

$$LC(e) < LC(e'') < LC(e'). \tag{24}$$

For example, for process p_1 there is an event, e_1^4, between the events e_1^3 and e_1^5 in Figure 8; indeed, $LC(e_1^3) = 3$, $LC(e_1^5) = 5$, $LC(e_1^4) = 4$, and $LC(e_1^3) < LC(e_1^4) < LC(e_1^5)$. However, for process p_3, the events e_3^3 and e_3^4 are consecutive though, $LC(e_3^3) = 3$ and $LC(e_3^4) = 10$.

2.7 Message delivery rules; causal delivery

The communication channel abstraction makes no assumptions about the order of messages; a real-life network might reorder messages. This fact has profound implications for a distributed application. Consider for example a robot getting instructions to navigate from a monitoring facility with two messages, "turn left" and "turn right", being delivered out of order.

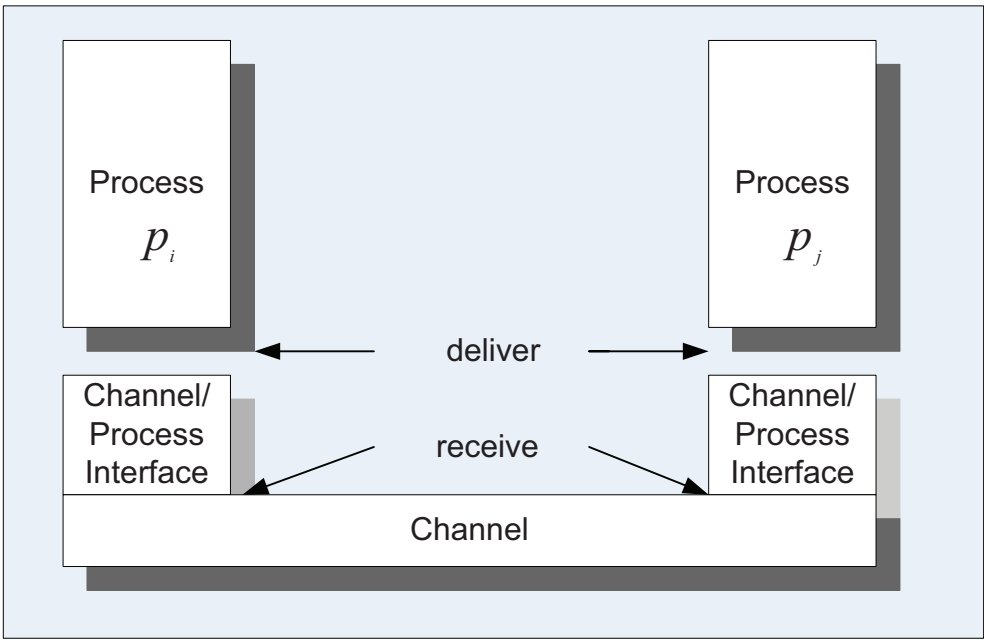

Figure 9: Message receiving and message delivery are two distinct operations. The channel-process interface implements the delivery rules, e.g., FIFO delivery.

Message receiving and message delivery are two distinct operations; a *delivery rule* is an additional assumption about the channel-process interface. This rule establishes when a message received is actually delivered to the destination process. The receiving of a message m and its delivery are two distinct events in a causal relation with one another, a message can only be delivered after being received, see Figure 9

$$receive(m) \rightarrow deliver(m). \tag{25}$$

First-In-First-Out (FIFO) delivery implies that messages are delivered in the same order they are sent. For each pair of source-destination processes (p_i, p_j) FIFO delivery requires that the following relation should be satisfied

$$send_i(m) \rightarrow send_i(m') \Rightarrow deliver_j(m) \rightarrow deliver_j(m'). \tag{26}$$

Even if the communication channel does not guarantee FIFO delivery, FIFO delivery can be enforced by attaching a sequence number to each message sent. The sequence numbers are also used to reassemble messages out of individual packets.

Causal delivery is an extension of the FIFO delivery to the case when a process receives messages from different sources. Assume a group of three processes, (p_i, p_j, p_k) and two messages m and m'. Causal delivery requires that

$$send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m'). \tag{27}$$

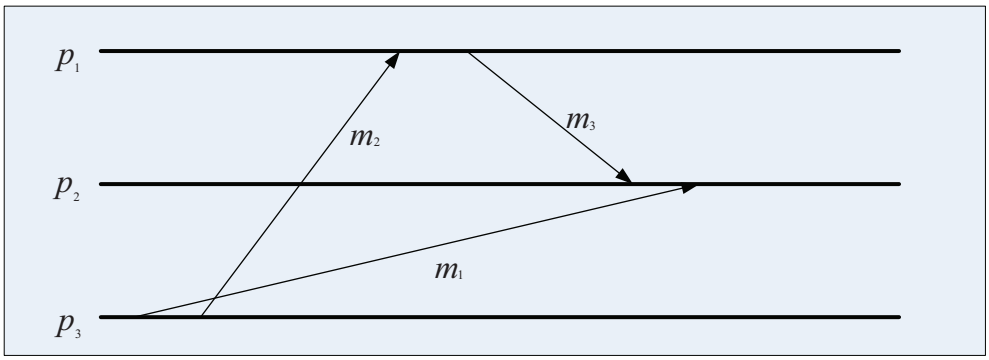

Figure 10: Violation of causal delivery when more than two processes are involved; message m_1 is delivered to process p_2 after message m_3, though message m_1 was sent before m_3. Indeed, message m_3 was sent by process p_1 after receiving m_2, which in turn was sent by process p_3 after sending message m_1.

When more than two processes are involved in a message exchange, the message delivery may be FIFO, but not causal as shown in Figure 10 where we see that

- $deliver(m_3) \rightarrow deliver(m_1)$; according to the local history of process p_2.
- $deliver(m_2) \rightarrow send(m_3)$; according to the local history of process p_1.
- $send(m_1) \rightarrow send(m_2)$; according to the local history of process p_3.
- $send(m_2) \rightarrow deliver(m_2)$.
- $send(m_3) \rightarrow deliver(m_3)$.

The transitivity property and the causality relations above imply that $send(m_1) \rightarrow deliver(m_3)$.

Call $TS(m)$ the *time stamp* carried by message m. A message received by process p_i is *stable* if no future messages with a time stamp smaller than $TS(m)$ can be received by

process p_i. When using logical clocks, a process p_i can construct consistent observations of the system if it implements the following delivery rule: *deliver all stable messages in increasing time stamp order.*

Let us now examine the problem of *consistent message delivery* under several sets of assumptions. First, assume that processes cooperating with each other in a distributed environment have access to a *global real-time clock*, that the message delays are bounded by δ, and that there is no clock drift. Call $RC(e)$ the time of occurrence of event e. A process includes in every message it sends $RC(e)$, where e is the send message event. The delivery rule in this case is: *at time t deliver all received messages with time stamps up to $(t-\delta)$ in increasing time stamp order.* Indeed, this delivery rule guarantees that under the bounded delay assumption the message delivery is consistent. All messages delivered at time t are in order and no future message with a time stamp lower than any of the messages delivered may arrive.

For any two events, e and e', occurring in different processes, the so called *clock condition* is satisfied if

$$e \rightarrow e' \Rightarrow RC(e) < RC(e'),\ \forall e, e'. \tag{28}$$

Oftentimes, we are interested in determining the set of events that caused an event knowing the time stamps associated with all events, in other words, we are interested in deducing the causal precedence relation between events from their time stamps. To do so we need to define the so-called strong clock condition. The *strong clock condition* requires an equivalence between the causal precedence and the ordering of the time stamps

$$\forall e, e', \quad e \ \rightarrow \ e' \ \equiv \ TS(e) \ < \ TS(e'). \tag{29}$$

Causal delivery is very important because it allows processes to reason about the entire system using only local information. This is only true in a closed system where all communication channels are known; sometimes the system has *hidden channels* and reasoning based on causal analysis may lead to incorrect conclusions.

2.8 Runs and cuts; causal history

Knowledge of the state of several, possibly all, processes in a distributed system is often needed. For example, a supervisory process must be able to detect when a subset of processes is deadlocked; a process might migrate from one location to another or be replicated only after an agreement with others. In all these examples a process needs to evaluate a predicate function of the global state of the system.

We call the process responsible for constructing the global state of the system, the *monitor*; a monitor sends messages requesting information about the local state of every process and gathers the replies to construct the global state. Intuitively, the construction of the global state is equivalent to taking snapshots of individual processes and then combining these snapshots into a global view. Yet, combining snapshots is straightforward if and only if all processes have access to a global clock and the snapshots are taken at the same time; hence, the snapshots are consistent with one another.

A *run* is a total ordering R of all the events in the global history of a distributed computation consistent with the local history of each participant process; a run

$$R = (e_1^{j_1}, e_2^{j_2}, \ldots, e_n^{j_n}) \tag{30}$$

implies a sequence of events as well as a sequence of global states.

For example, consider the three processes in Figure 11. We can construct a three-dimensional lattice of global states following a procedure similar to the one in Figure 5 starting from the initial state $\Sigma^{(000)}$ and proceeding to any reachable state $\Sigma^{(ijk)}$ with i, j, k the events in processes p_1, p_2, p_3, respectively. The run $R_1 = (e_1^1, e_2^1, e_3^1, e_1^2)$ is consistent with both the local history of each process and the global history; this run is valid, the system has traversed the global states

$$\Sigma^{000}, \Sigma^{100}, \Sigma^{110}, \Sigma^{111}, \Sigma^{211} \tag{31}$$

On the other hand, the run $R_2 = (e_1^1, e_1^2, e_3^1, e_1^3, e_3^2)$ is invalid because it is inconsistent with the global history. The system cannot ever reach the state Σ^{301}; message m_1 must be sent before it is received, so event e_2^1 must occur in any run before event e_1^3.

A *cut* is a subset of the local history of all processes. If h_i^j denotes the history of process p_i up to and including its j-th event, e_i^j, then a cut C is an n-tuple

$$C = \{h_i^j\} \quad \text{with} \quad i \in \{1, n\} \text{ and } j \in \{1, n_i\}. \tag{32}$$

The frontier of the cut is an n-tuple consisting of the last event of every process included in the cut. Figure 11 illustrates a *space-time diagram* for a group of three processes, p_1, p_2, p_3 and it shows two cuts, C_1 and C_2. C_1 has the frontier $(4, 5, 2)$, frozen after the fourth event of process p_1, the fifth event of process p_2 and the second event of process p_3, and C_2 has the frontier $(5, 6, 3)$.

Cuts provide the necessary intuition to generate global states based on an exchange of messages between a monitor and a group of processes. The cut represents the instance when requests to report individual state are received by the members of the group. Clearly not all cuts are meaningful. For example, the cut C_1 with the frontier $(4, 5, 2)$ in Figure 11 violates our intuition regarding causality; it includes e_2^4, the event triggered by the arrival of message m_3 at process p_2 but does not include e_3^3, the event triggered by process p_3 sending m_3. In this snapshot p_3 was frozen after its second event, e_3^2, before it had the chance to send message m_3. Causality is violated and the system cannot ever reach such a state.

Next we introduce the concepts of consistent and inconsistent cuts and runs. A cut closed under the *causal precedence relationship* is called a *consistent cut*. C is a consistent cut if and only if for all events

$$\forall e, e', \quad (e \in C) \ \wedge \ (e' \ \rightarrow \ e) \ \Rightarrow \ e' \in C. \tag{33}$$

A consistent cut establishes an "instance" of a distributed computation; given a consistent cut we can determine if an event e occurred before the cut.

A run R is said to be consistent if the total ordering of events imposed by the run is consistent with the partial order imposed by the causal relation; for all events, $e \rightarrow e'$ implies that e appears before e' in R.

Consider a distributed computation consisting of a group of communicating processes $G = \{p_1, p_2, ..., p_n\}$. The *causal history of event* $e, \gamma(e)$, is the smallest consistent cut of G including event e

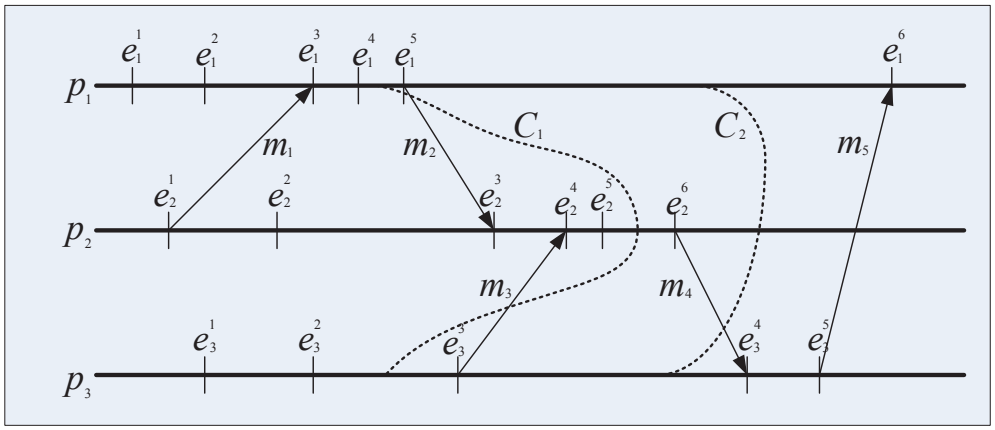

Figure 11: Inconsistent and consistent cuts: the cut $C_1 = (e_1^4, e_2^5, e_3^2)$ is inconsistent because it includes e_2^4, the event triggered by the arrival of the message m_3 at process p_2, but does not include e_3^3, the event triggered by process p_3 sending m_3 thus, the cut C_1 violates causality. On the other hand, $C_2 = (e_1^5, e_2^6, e_3^3)$ is a consistent cut, there is no causal inconsistency, it includes event e_2^6, the sending of message m_4, without the effect of it, the event e_3^4 receiving the message by process p_3.

$$\gamma(e) = \{e' \in G \mid e' \rightarrow e\} \ \cup \ \{e\}. \tag{34}$$

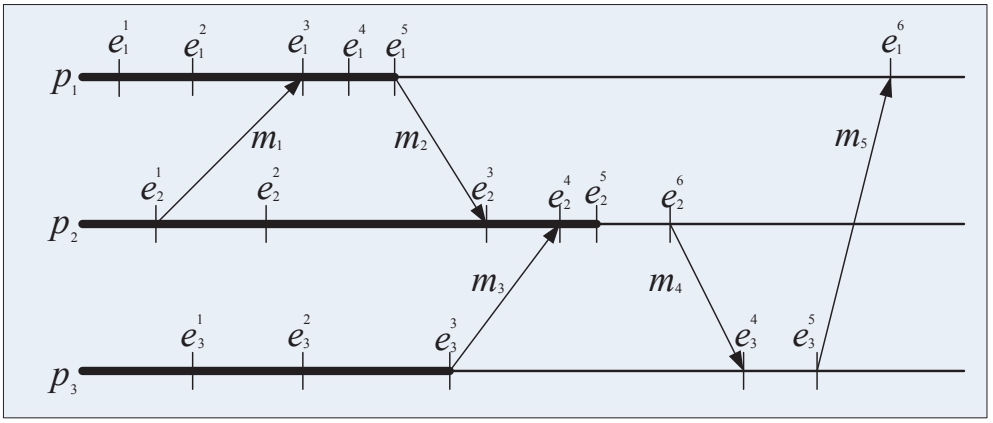

Figure 12: The causal history of event e_2^5, $\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^3, e_3^3\}$, is the smallest consistent cut including e_2^5.

The causal history of event e_2^5 in Figure 12 is:

$$\gamma(e_2^5) = \{e_1^1, e_1^2, e_1^3, e_1^4, e_1^5, e_2^1, e_2^2, e_2^3, e_2^4, e_2^5, e_3^1, e_3^3, e_3^3\}. \tag{35}$$

This is the smallest consistent cut including e_2^5; indeed, if we omit e_3^3, then the cut $(5, 5, 2)$ would be inconsistent, it would include e_2^4, the communication event for receiving m_3, but not e_3^3, the sending of m_3. If we omit e_1^5, the cut $(4, 5, 3)$ would also be inconsistent, it would include e_2^3 but not e_1^5.

Causal histories can be used as clock values and satisfy the strong clock condition provided that we equate clock comparison with set inclusion. Indeed.

$$e \rightarrow e' \ \equiv \ \gamma(e) \subset \gamma(e'). \tag{36}$$

The following algorithm can be used to construct causal histories:

- Each $p_i \in G$ starts with $\theta = \emptyset$.
- Every time p_i receives a message m from p_j it constructs

$$\gamma(e_i) \ = \ \gamma(e_j) \cup \gamma(e_k) \tag{37}$$

  with e_i the *receive* event, e_j the previous local event of p_i, e_k the *send* event of process p_j.

Unfortunately, this concatenation of histories is impractical because the causal histories grow very fast.

Now we present a protocol to construct consistent global states based on the monitoring concepts discusses in this section. We assume a fully connected network; recall that given two processes p_i and p_j, the state $\xi_{i,j}$ of the channel from p_i to p_j consists of messages sent by p_i but not yet received by p_j. The snapshot protocol of Chandy and Lamport consists of three steps [72]

1. Process p_0 sends to itself a "take snapshot" message.
2. Let p_f be the process from which p_i receives the"take snapshot" message for the first time. Upon receiving the message, the process p_i records its local state, σ_i, and relays the "take snapshot" along all its outgoing channels without executing any events on behalf of its underlying computation; channel state $\xi_{f,i}$ is set to empty and process p_i starts recording messages received over each of its incoming channels.
3. Let p_s be the process from which p_i receives the "take snapshot" message beyond the first time; process p_i stops recording messages along the incoming channel from p_s and declares channel state $\xi_{s,i}$ as those messages that have been recorded.

Each"take snapshot" message crosses each channel exactly once and every process p_i has made its contribution to the global state; a process records its state the first time it receives a "take snapshot" message and then stops executing the underlying computation for some time. Thus, in a fully connected network with n processes the protocol requires $n \times (n-1)$ messages, one on each channel.

For example, consider a set of six processes, each pair of processes being connected by two unidirectional channels as shown in Figure 13. Assume that all channels are empty, $\xi_{i,j} = 0, \ i \in \{0, 5\}, \ j \in \{0, 5\}$, at the time when process p_0 issues the "take snapshot" message. The actual flow of messages is

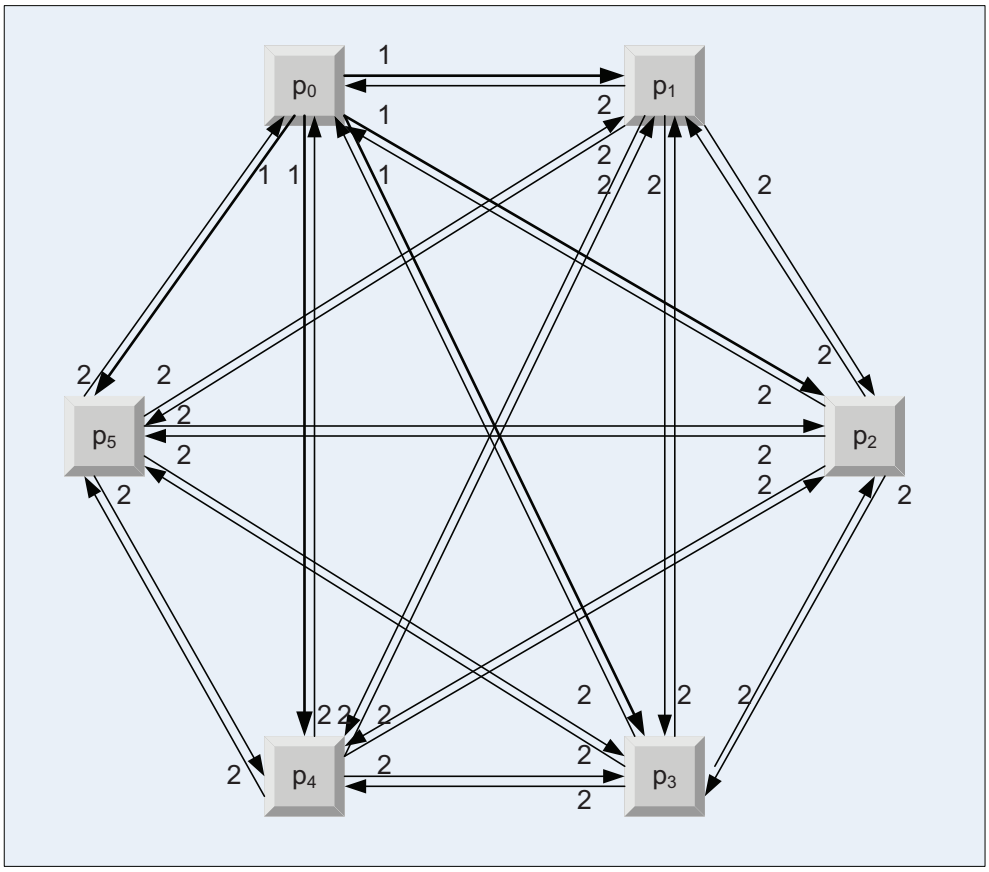

Figure 13: Six processes executing the snapshot protocol.

- In step 0, p_0 sends to itself the "take snapshot" message.
- In step 1, process p_0 sends five "take snapshot" messages labeled (1) in Figure 13.
- In step 2, each of the five processes, p_1, p_2, p_3, p_4, and p_5 sends a "take snapshot" message labeled (2) to every other process.

A "take snapshot" message crosses each channel from process p_i to p_j, $i, j \in \{0, 5\}$ exactly once and $6 \times 5 = 30$ messages are exchanged.

2.9 Concurrency

Concurrency means that several activities are executed simultaneously. Concurrency allows us to reduce the execution time of a data-intensive problem as discussed in Section 2.1. To exploit concurrency often we have to take a fresh look at the problem and design a parallel algorithm. In other instances we can still use the sequential algorithm in the context of the SPMD paradigm.

Concurrency is a critical element of the design of system software. The kernel of an operating system exploits concurrency for virtualization of system resources such as the processor and the memory. *Virtualization*, covered in depth in Chapter 5.1, is a system design strategy with a broad range of objectives including:

- Hiding latency and performance enhancement, e.g., schedule a ready-to-run thread when the current thread is waiting for the completion of an I/O operation;
- Avoiding limitations imposed by the physical resources, e.g., allow an application to run in a virtual address space of a standard size, rather than be restricted by the physical memory available on a system;
- Enhancing reliability and performance, as in the case of RAID systems mentioned in Section 3.5.

Sometimes concurrency is used to describe activities that appear to be executed simultaneously, though only one of them may be active at any given time, as in the case of processor virtualization when multiple threads appear to run concurrently on a single processor. A thread can be suspended due to an external event and a context switch to a different thread takes place. The state of the first thread is saved and the state of another thread ready to proceed is loaded and the thread is activated. The suspended thread will be re-activated at a later point in time.

Dealing with some of the effects of concurrency can be very challenging. Context switching could involve multiple components of a OS kernel including the Virtual Memory Manager (VMM), the Exception Handler (EH), the Scheduler (S), and the Multi-level Memory Manager (MLMM). When a page fault occurs during the fetching of the next instruction multiple context switches are necessary as shown in Figure 14.

Concurrency is often motivated by the desire to enhance the system performance. For example, in a pipelined computer architecture multiple instructions are in different phases of execution at any given time. Once the pipeline is full, a result is produced at every pipeline cycle; an n-stage pipeline could potentially lead to a speedup by a factor of n. There is always a price to pay for increased performance and in this example is design complexity and cost. An n-stage pipeline requires n execution units, one for each stage, as well as a coordination unit. It also requires a careful timing analysis in order to achieve the full speed-up.

This example shows that the management and the coordination of the concurrent activities increases the complexity of a system. The interaction between pipelining and virtual memory further complicates the functions of the kernel; indeed, one of the instructions in the pipeline could be interrupted due to a page fault and the handling of this case requires special precautions, as the state of the processor is difficult to define.

While in the early days of computing concurrency was analyzed mostly in the context of the system software, nowadays concurrency is an ubiquitous feature of many applications. *Embedded* systems are a class of concurrent systems used only by the critical infrastructure, but also by the most diverse systems from ignition in a car to oil processing in a refinery, from smart meeters to coffee makers. Embedded controllers for reactive real-time applications are implemented as mixed software-hardware systems [293].

Concurrency is exploited by application software to speedup a computation and to allow a number of clients to access a service. Parallel applications partition the workload and distribute it to multiple threads running concurrently. Distributed applications, including transaction management systems and applications based on the client-server paradigm discussed in Section 2.13, use extensively concurrency to improve the response time. For example, a web server spawns a new thread when a new request is received thus, multiple

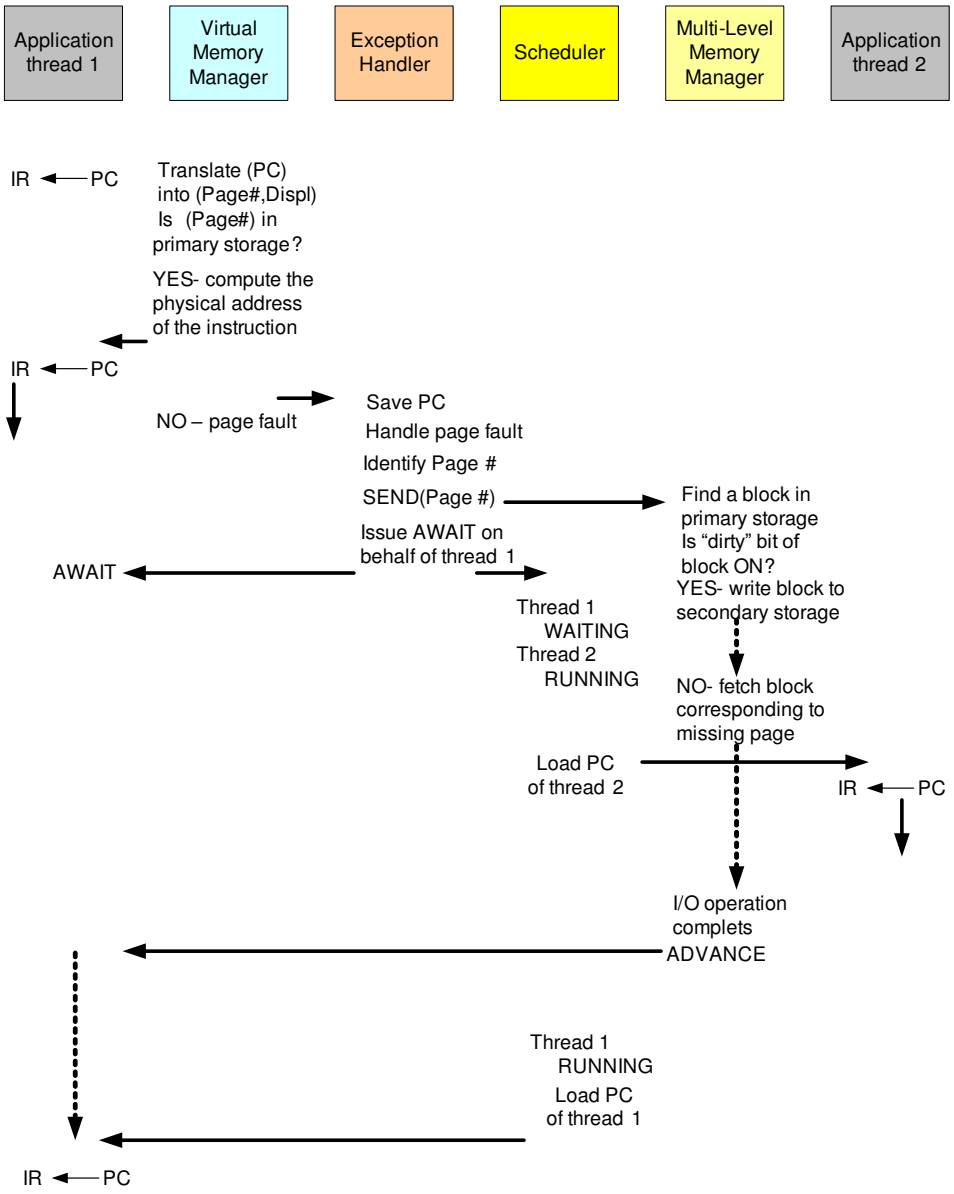

Figure 14: Context switching when a page fault occurs during the instruction fetch phase. VMM attempts to translate the virtual address of a next instruction of thread 1 and encounters a page fault. Then thread 1 is suspended waiting for an event when the page is brought in the physical memory from the disk. The Scheduler dispatches thread 2. To handle the fault the EX invokes the MLMM.

server threads run concurrently. A main attraction for hosting Web-based applications is the cloud elasticity, the ability of a service running on a cloud to acquire resources as needed and to pay for these resources as they are consumed.

Communication channels allow concurrent activities to work in concert and coordinate.

Communication protocols allow us to transform noisy and unreliable channels into reliable ones which deliver messages in order. As mentioned earlier, concurrent activities communicate with one another via shared memory or via message passing. Multiple instances of a cloud application, a server and the clients of the service it provides, and many other applications communicate via message passing. The Message Passing Interface (MPI) supports both synchronous and asynchronous communication and it is often used by parallel and distributed applications. Message passing enforces modularity, as we have seen in Section 2.13, and prevents the communicating activities from *sharing their fate*; a server could fail without affecting the clients which did not use the service during the period the server was unavailable.

The communication patterns in case of a parallel application are more structured, while patterns of communication for concurrent activities of a distributed application are more dynamic and unstructured. Barrier synchronization requires the threads running concurrently to wait until all of them have completed the current task before proceeding to the next. Sometimes, one of the activities, a coordinator, mediates communication among concurrent activities, in other instances individual threads communicate directly with one another.

2.10 Atomic actions

Parallel and distributed applications must take special precautions for handling shared resources. For example, consider a financial application where the shared resource is an account record; a thread running on behalf of a transaction first accesses the account to read the current balance, then updates the balance, and, finally, writes back the new balance. When a thread is interrupted before being able to complete the three steps of the process the results of the financial transactions are incorrect if another thread operating on the same account is allowed to proceed. Another challenge is to deal with a transaction involving the transfer from one account to another. A system crash after the completion of the operation on the first account will again lead to an inconsistency, the amount debited from the first account is not credited to the second.

In these cases, as in many other similar situations, a multi-step operation should be allowed to proceed to completion without any interruptions, the operation should be *atomic*. An important observation is that such atomic actions should not expose the state of the system until the action is completed. Hiding the internal state of an atomic action reduces the number of states a system can be in thus, it simplifies the design and maintenance of the system. An atomic action is composed of several steps and each one of them may fail; therefore, we have to take additional precautions to avoid exposing the internal state of the system in case of such a failure.

The discussion of the transaction system suggests that an analysis of atomicity should pay special attention to the basic operation of updating the value of an object in storage. Even to modify the contents of a memory location several machine instructions must be executed: load the current value in a register, modify the contents of the register, and store back the result.

Atomicity cannot be implemented without some hardware support; indeed, the instruction set of most processors support the *Test-and-Set* instruction which writes to a memory location and returns the old content of that memory cell as non-interruptible operations; other architectures support *Compare-and-Swap*, an atomic instruction which compares the

contents of a memory location to a given value and, only if the two values are the same, modifies the contents of that memory location to a given new value.

Two flavors of atomicity can be distinguished: *all-or-nothing* and *before-or-after* atomicity. *All-or-nothing* means that either the entire atomic action is carried out, or the system is left in the same state it was before the atomic action was attempted; in our examples a transaction is either carried out successfully, or the record targeted by the transaction is returned to its original state. The states of an *all-or-nothing* action are shown in Figure 15.

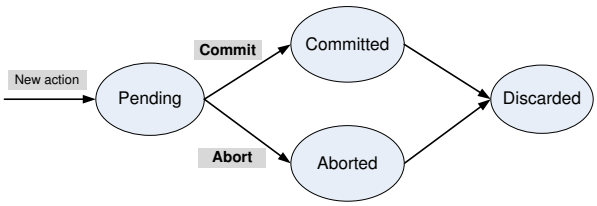

Figure 15: The states of an *all-or-nothing* action.

To guarantee the all-or-nothing property of an action we have to distinguish preparatory actions which can be undone from irreversible ones, such as the alteration of the only copy of an object. Such preparatory actions are: allocation of a resource, fetching a page from secondary storage, allocation of memory on the stack, and so on. One of the golden rules of data management is never to change the only copy; maintaining the history of changes and a log of all activities allow us to deal with system failures and to ensure consistency.

An all-or-nothing action consists of a *pre-commit* and a *post-commit* phase; during the former it should be possible to back up from it without leaving any trace while the later phase should be able to run to completion. The transition from the first to the second phase is called a *commit point*. During the *pre-commit* phase all steps necessary to prepare the post-commit phase, e.g., check permissions, swap in main memory all pages that may be needed, mount removable media, and allocate stack space must be carried out; during this phase no results should be exposed and no actions that are irreversible should be carried out. Shared resources allocated during the pre-commit cannot be released until after the commit point. The commit step should be the last step of an all-or-nothing action.

A discussion of storage models illustrates the effort required to support all-or-nothing atomicity, see Figure 16. The common storage model implemented by hardware is the so called *cell storage*, a collection of cells each capable to hold an object, e.g., the primary memory of a computer where each cell is addressable. Cell storage does not support all-or-nothing actions, once the contents of a cell is changed by an action, there is no way to abort the action and restore the original content of the cell.

To be able to restore a previous value we have to maintain a *version history* for each variable in the cell storage. The storage model that supports all-or-nothing actions is called *journal storage*. Now the cell storage is no longer accessible to the action by the access is mitigated by a *storage manager*. In addition to the basic primitives to *Read* an existing value and to *Write* a new value in cell storage, the storage manager uniquely identifies an action that changes the value in cell storage and, when the action is aborted is able to retrieve the version of the variable before the action and restore it. When = the action is committed then the new value should be written to the cell.

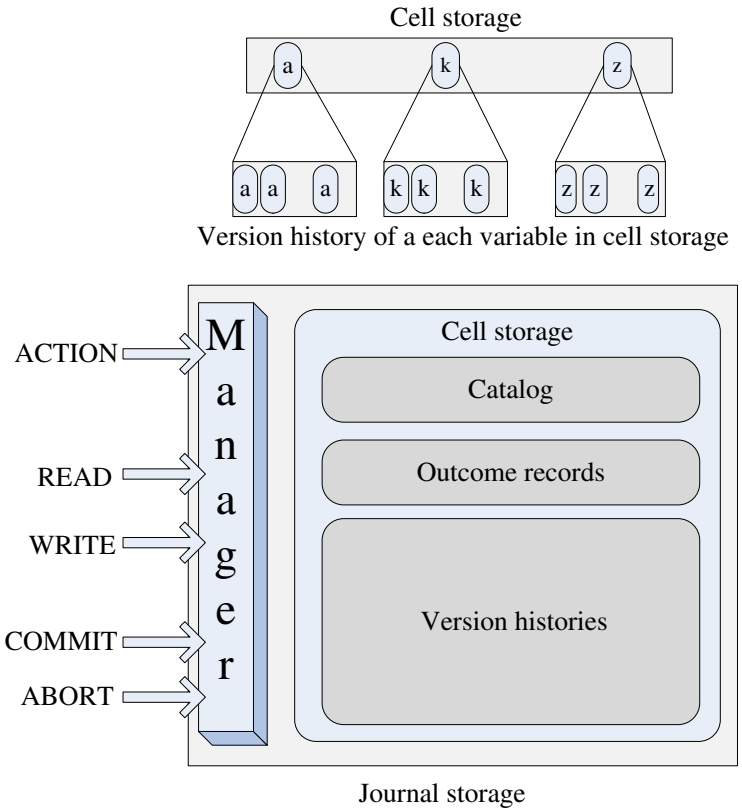

Figure 16: Storage models. Cell storage does not support all-or-nothing actions. When we maintain the version histories it is possible to restore the original content but we need to encapsulate the data access and provide mechanisms to implement the two phases of of an atomic all-or-nothing action. The journal storage does precisely that.

Figure 16 shows that for a journal storage in addition to the version histories of all variables affected by the action we have to implement a catalog of variables and also to maintain a record to identify each new action. A new action first invokes the *Action* primitive; at that time an outcome record uniquely identifying the action is created. Then, every time the action accesses a variable, the version history is modified and, finally, the action either invokes a *Commit* or an *Abort* primitive. In the journal storage model the action is atomic and follows the state transition diagram in Figure 15.

Before-or-after atomicity means that, from the point of view of an external observer, the effect of multiple actions is as if these actions have occurred one after another, in some order; a stronger condition is to impose a sequential order among transitions. In our example the transaction acting on two accounts should either debit the first account and then credit the second one, or leave both accounts unchanged. The order is important, as the first account cannot be left with a negative balance.

Atomicity is a critical concepts for our efforts to build reliable systems from unreliable

components and, at the same time, to support as much parallelism as possible for better performance. Atomicity allows us to deal with unforseen events and to support coordination of concurrent activities. The unforseen event could be a system crash, a request to share a control structure, the need to suspend an activity, and so on; in all these cases we have to save the state of the process or of the entire system to be able to restart it at a later time.

As atomicity is required in many contexts, it is desirable to have a systematic approach rather than an ad-hoc one. A systematic approach to atomicity must address several delicate questions:

- How to guarantee that only one atomic action has access to a shared resource at any given time.
- How to return to the original state of the system when an atomic action fails to complete.
- How to ensure that the order of several atomic actions leads to consistent results.

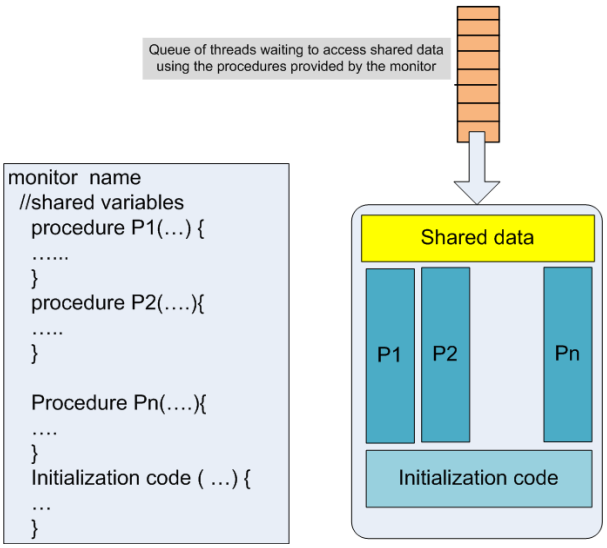

Figure 17: A monitor provides special procedures to access the data in a critical section.

Answers to these questions increase the complexity of the system and often generate additional problems. For example, access to shared resources can be protected by locks, but when there are multiple shared resources protected by locks concurrent activities may deadlock. A *lock* is a construct which enforces sequential access to a shared resource; such actions are packaged in the *critical sections* of the code. If the lock is not set, a thread first locks the access, then enters the critical section and finally unlocks it; a thread wishing to enter the critical section finds the lock set and waits for the lock to be reset. A lock can be implemented using the hardware instructions supporting atomicity.

Semaphores and monitors are more elaborate structures ensuring serial access; semaphores force processes to queue when the lock is set and are released from this queue and allowed

to enter the critical section one by one. Monitors provide special procedures to access the shared data see Figure 17. *The mechanisms for the process coordination we described require the cooperation of all activities,* the same way traffic lights prevent accidents only as long as the drivers follow the rules.

2.11 Consensus protocols

Consensus is a pervasive problem in many areas of human endeavor; consensus is the process of agreeing to one of several alternates proposed by a number of agents. We restrict our discussion to the case of a distributed system when the agents are a set of processes expected to reach consensus on a single proposed value.

No fault-tolerant consensus protocol can guarantee progress [123], but protocols which guarantee freedom from inconsistencies (safety) have been developed. A family of protocols to reach consensus based on a finite state machine approach is called *Paxos*[10].

A fair number of contributions to the family of Paxos protocols are discussed in the literature. Leslie Lamport has proposed several versions of the protocol including *Disk Paxos, Cheap Paxos, Fast Paxos, Vertical Paxos, Stoppable Paxos, Byzantizing Paxos by Refinement, Generalized Consensus and Paxos* and *Leaderless Byzantine Paxos*; he has also published a paper on the fictional part-time parliament in Paxos [206] and a layman's dissection of the protocol [207].

The *consensus service* consists of a set of n processes; *clients* send requests to processes and propose a value and wait for a response; the goal is to get the set of processes to reach consensus on a single proposed value. The *basic Paxos* protocol is based on several assumptions about the processors and the network:

- The processes run on processors and communicate through a network; the processors and the network may experience failures, but not Byzantine failures[11].
- The processors: (i) operate at arbitrary speeds; (ii) have stable storage and may rejoin the protocol after a failure; (iii) can send messages to any other processor.
- The network: (i) may lose, reorder, or duplicate messages; (ii) messages are sent asynchronously and may take arbitrary long time to reach the destination.

The *basic Paxos* considers several types of entities: (a) *client,* an agent that issues a request and waits for a response; (b) *proposer,* an agent with the mission to advocate a request from a client, convince the acceptors to agree on the value proposed by a client, and to act as a coordinator to move the protocol forward in case of conflicts; (c) *acceptor*,

[10]Paxos is a small Greek island in the Ionian Sea; a fictional consensus procedure is attributed to an ancient Paxos legislative body. The island had a part-time parliament as its inhabitants were more interested in other activities than in civic work; "the problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by todays fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing" according to Leslie Lamport [206] (for additional papers see http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html).

[11]A Byzantine failure in a distributed system could be an *omission failure*, e.g., a crash failure, failure to receive a request or to send a response; it could also be a *commission failure*, e.g., process a request incorrectly, corrupt the local state, and/or send an incorrect or inconsistent response to a request.

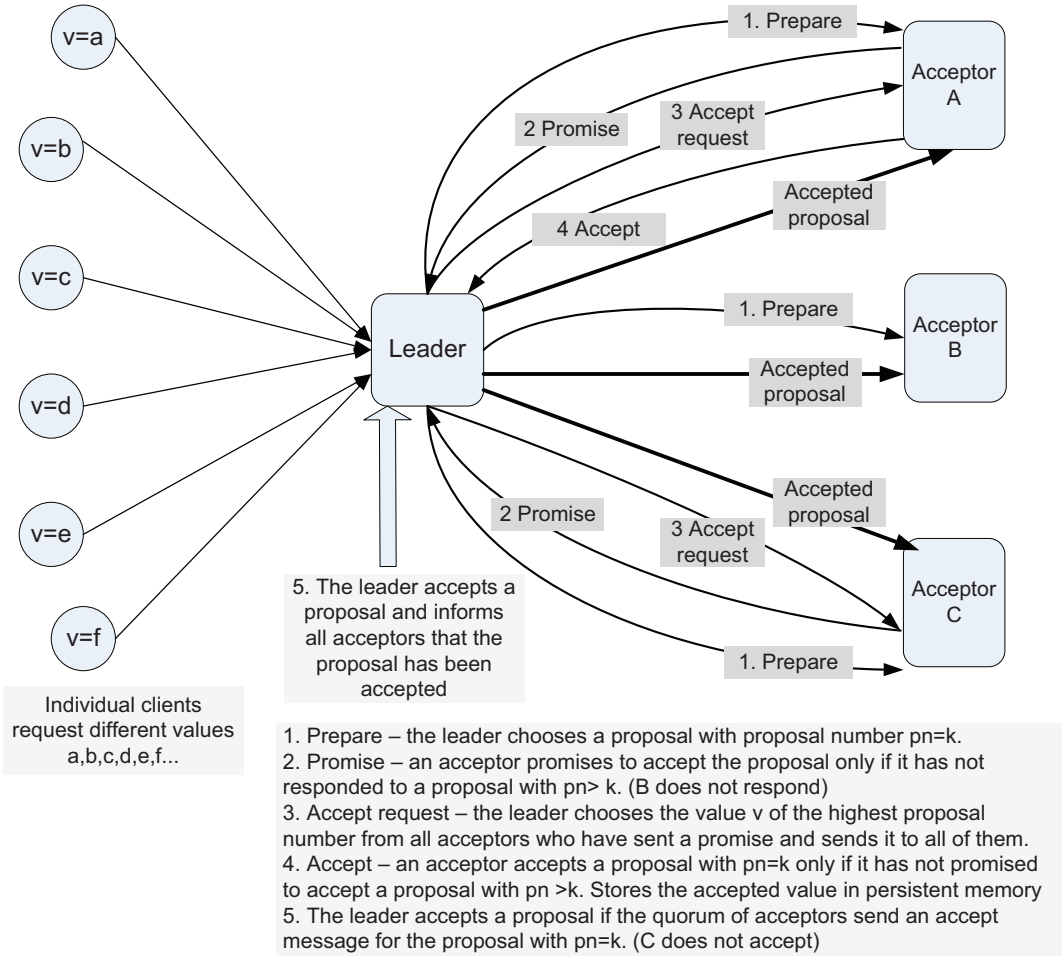

Figure 18: The flow of messages for the Paxos consensus algorithm. Individual clients propose different values to the leader who initiates the algorithm. Acceptor A accepts the value in message with proposal number pn=k; acceptor B does not respond with a promise while acceptor C responds with a promise, but ultimately does not accept the proposal.

an agent acting as the fault-tolerant "memory" of the protocol; (d) *learner,* an agent acting as the replication factor of then protocol and taking action once a request has been agreed upon; and finally (e) the *leader,* a distinguished proposer.

A *quorum* is a subset of all acceptors. A proposal has a proposal number pn and contains a value v. Several types of requests flow through the system, *prepare, accept.*

In a typical deployment of the algorithm an entity plays three roles, proposer, acceptor, and learner. Then the flow of messages can be described as follows [207]: "clients send messages to a leader; during normal operations the leader receives the client's command, assigns it a new command number i, and then begins the i-th instance of the consensus algorithm by sending messages to a set of acceptor processes." By merging the roles, the protocol "collapses" into an efficient client-master-replica style protocol.

A proposal consists of a pair, a unique proposal number and a proposed value, (pn, v); multiple proposals may propose the same value v. A value is chosen if a simple majority of acceptors have accepted it. We need to guarantee that at most one value can be chosen,

otherwise there is no consensus. The two phases of the algorithm are:

Phase I.

1. *Proposal preparation:* a proposer (the leader) sends a proposal ($pn = k, v$). The proposer chooses a proposal number $pn = k$ and sends a *prepare message* to a majority of acceptors requesting:
   - that a proposal with $pn < k$ should not be accepted;
   - the $pn < k$ of the highest number proposal already accepted by each acceptor.
2. *Proposal promise:* An acceptor must remember the proposal number of the highest proposal number it has ever accepted as well as the highest proposal number it has ever responded to. The acceptor can accept a proposal with $pn = k$ if and only if it has not responded to a prepare request with $pn > k$; if it has already replied to a prepare request for a proposal with $pn > k$ then it should not reply. Lost messages are treated as an acceptor that chooses not to respond.

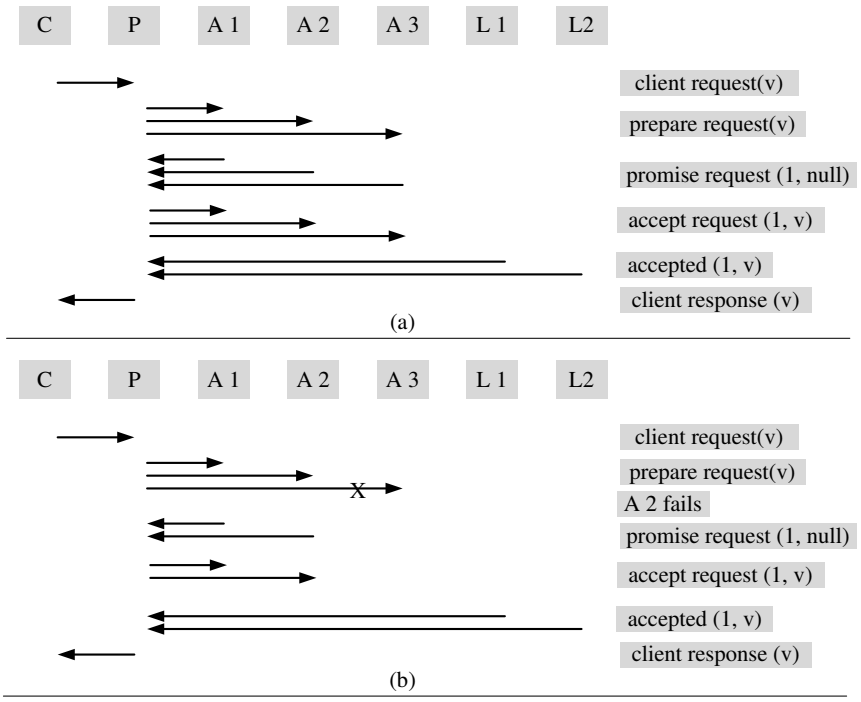

Figure 19: The basic Paxos with three actors: proposer (P), three acceptors (A1, A2, A3), and two learners (L1, L2). The client (C) sends a request to one of the actors playing the role of a proposer. The entities involved are (a) Successful first round when there are no failures. (b) Successful first round of Paxos when an acceptor fails.

Phase II.

1. *Accept request:* if the majority of acceptors respond, then the proposer chooses the value v of the proposal as follows:

- the value v of the highest proposal number selected from all the responses;
- an arbitrary value if no proposal was issued by any of the proposers.

The proposer sends an *accept request* message to a quorum of acceptors including $(pn = k, v)$

2. *Accept:* If an acceptor receives an *accept message* for a proposal with the proposal number $pn = k$ it must accept it if and only if it has not already promised to consider proposals with a $pn > k$. If it accepts the proposal it should register the value v and send an *accept* message to the proposer and to every learner; if it does not accept the proposal it should ignore the request.

The following properties of the algorithm are important to show its correctness: (1) a proposal number is unique; (2) any two sets of acceptors have at least one acceptor in common; and (3) the value sent out in Phase 2 of the algorithm is the value of the highest numbered proposal of all the responses in Phase 1.

Figure 18 illustrates the flow of messages for the consensus protocol. A detailed analysis of the message flows for different failure scenarios and of the properties of the protocol can be found in [207]. We only mention that the protocol defines three safety properties: (1) non-triviality - the only values that can be learned are proposed values; (2) consistency - at most one value can be learned; and (3) liveness - if a value v has been proposed eventually every learner will learn some value, provided that sufficient processors remain non-faulty. Figure 19 shows the message exchange when there are three actors involved.

In Section 4.5 we present a consensus service, the *ZooKeeper*, based on the Paxos protocol and in Section 8.6 we discuss *Chubby*, a locking service based on the algorithm.