Distributed System Models

1. Introduction: the need for models

These notes introduce the main models of distributed systems: shared memory and, in particular, message passing. In general, a model contains only the essential ingredients that we need to consider in order to understand and reason about a system’s behaviour. A system model has to address the questions:

• What are the main entities in the system?
• How do they interact?
• What are their characteristics, that affect their individual and collective behaviour?

The purpose of a model is:

• To make explicit all our assumptions about the systems we are modelling
• To make generalisations (theorems) concerning what is possible or impossible, given those assumptions.

Consider the design of a distributed file system such as NFS. There are many factors that we would immediately say the design depends upon. For example, if a server computer disobeyed the protocol and sent garbage messages as its responses, or if it responded only after waiting an hour, the file system would break down. Equally, it is not hard to identify some features that the design does not depend on, at least in its essential respects. For example, it does not depend on whether the computers have CISC or RISC processors, on whether the computers are connected by an Ethernet or token-ring network, on whether the fields in the standard message headers have any particular order, or on whether any individual message is dropped by the network.

There is much to be gained by knowing what our designs do and do not depend upon. It allows us to decide whether a design will work if we try to implement it in a particular system: we need only ask whether our assumptions hold in that system. Also, by making our assumptions clear and explicit, we can hope to prove system properties using mathematical techniques. These properties will then hold for any system meeting our assumptions. Finally, by abstracting only the essential system entities and characteristics away from details such as hardware, we can clarify our understanding of our systems.

A little thought tells us that a model of a distributed system must include:

• Processes
• How they communicate
• The timing assumptions
• How processes, communication channels and other components may fail.

Note that already there is no mention of machines or networks. The process concept combines activity and memory protection, which are the essential features of computers. Communication channels abstract away from physical communication media.

We make the following general characterisation of processes in a distributed system:
They have their own state, which is abstract data that they can transform. In general, they may also access and manipulate real-world objects such as devices in a factory.

In the message passing model processes interact only by sending and receiving messages and they do not share memory; in the memory-sharing model, they share a specified region of memory and they interact by reading and writing shared variables.

When considering a process’s interactions with the outside world (other processes or devices), it is irrelevant whether the process is internally single-threaded or multi-threaded.

In the remainder of these notes Section 2 distinguishes between message passing and shared memory models, and defines the main properties to be considered in constructing message passing models. In Section 3 onward we go on to outline some fundamental problems, proofs and results.

Note that although we are referring to distributed ‘systems’ in these notes, the processes under consideration may in fact belong to some executing application: at the level of our models we are making no distinction between these notions.

2. Properties of models

A distributed system contains a collection of processes \( p_i, i = 1, \ldots, n \). We can split models into two broad categories according to how processes interact: message passing and shared memory.

2.1 The message passing model

We have already considered message-based models in Section 10.3 of CDK (2nd. edition of Distributed Systems – Concepts and Design by Coulouris, Dollimore and Kindberg). Each process may engage in one of two types of communication event:

\[ \text{send}_i(m) - p_i \text{ sends the message } m \]
\[ \text{rcv}_j(m) - p_j \text{ receives the message } m \]

The message is transmitted over a communication channel connecting \( p_i \) and \( p_j \). There is a channel (or link) between each pair of processes that need to communicate. Each channel may be uni- or bi-directional. So the set of processes and channels forms a directed graph, with the processes as nodes and the channels edges.

We need not concern ourselves with the details of the network or networks used to implement the channels, any more than we consider the architecture of the processors used to implement the processes. But we are interested in abstract channel properties such as their reliability.

2.2 The shared memory model

Processes that are connected by a network may not access one another’s physical memory directly. However, CDK Chapter 17 describes the design and implementation of distributed shared memory, which is an abstraction of a repository of data objects that processes share and can read and/or update. Intuitively, we may implement distributed shared memory using a run-time system which locally traps each process’s accesses to the shared memory and which uses message passing (transparently to the process) to fetch the needed values or update other processes’ local copies of the shared data.

At its simplest, the programmer’s view of shared memory is as a collection of variables that each process may read or write. The two operations available are:

\[ R(x)u - \text{this denotes a read operation on the variable } x, \text{ which returns a value } u; \]
\[ W(y)v - \text{this is a write operation that sets the value of variable } y \text{ to be } v. \]

\(^1\)It is worth emphasising that when we say that \( p_i \) receives \( m \) we mean that the application takes delivery of the message. When we discuss message ordering in Chapter 11 we distinguish the event of receiving a message from that of delivering it to the application.
Just as processes in the message passing model can only interact with one another through the `send` and `rcv` operations, processes in this simple shared memory model may only affect one another through the `R` (read) and `W` (write) operations.

We may generalise the shared memory to consist of shared `objects`, which export abstract operations rather than just `read` and `write`. For example, two processes could share a `queue` object, which supports just the operations `enqueueAtTail` and `dequeueHead`. Processes are prohibited from accessing the queue except through these operations.

Both the message passing and shared memory models have received much practical and theoretical attention. There is no overriding reason to prefer one model to the other. We defer study of the shared memory model until CDK Chapter 17, and concentrate for now on message passing.

### 2.3 Processes, states and events

A process $p_i$ contains its own state, $s_i$. The question of how this state is represented lies outside our general model, but we can think of the state as consisting of variables in its address space, and perhaps the state of an attached device (for example, a robot arm). No other process may access the state. For the sake of definiteness, we eliminate all communication channels that are not part of our message passing model, by assuming that processes with attached devices cannot communicate through the outside world. For example, if a process controls a robot arm then no other process may use a sensor to observe the state of the robot arm or any effects of the robot arm.

As each process executes it takes a series of steps. A step is either a `send` or `rcv` operation, or it is an internal operation on its state. Leading on from the treatment of logical time in CDK Chapter 10, we characterise the `history` of process $p_i$ as a series of events corresponding to these steps:

$$ \text{history}(p_i) = h_i = e_i^1e_i^2e_i^3 \ldots $$

We are normally interested in all of a process’s communication actions, but we need not for now specify which internal events are of significance. The significance of a step will of course depend on the associated state of the process.

When process $p_i$ processes its $k$’th event $e_i^k$, the resultant local state is $s_i^k$. We can put the series of events and states in 1–1 correspondence,

$$ e_i^1 \leftrightarrow s_i^1, \quad e_i^2 \leftrightarrow s_i^2, \ldots $$

– starting from an initial state $s_i^0$, before $p_i$ has taken any steps.

Often we are interested only in an initial prefix of a process’s history, and write:

$$ h_i^k = e_i^1e_i^2 \ldots e_i^k \quad (k = 1, 2, \ldots). $$

We can also consider the `global history` of the system as the union of individual process histories,

$$ H = h_1 \cup h_2 \cup \ldots \cup h_n. $$

Note that we write this as a set, without any ordering imposed on the global events. A `run` is a total ordering of all the events in a global history, which is consistent with each local history’s ordering. From CDK Chapter 10 we also know that there is a partial ordering ‘→’ on $H$. A `linearisation` or consistent `run` is an ordering of the events in a global history that is consistent with this `happened-before` relation. Note that a linearisation is also a run. Run and linearisation orderings do not necessarily correspond to the chronological order in which events occurred in some execution of the system, should this be known. Indeed, two events may occur exactly simultaneously in some executions.

It is straightforward to observe the succession of states of an individual process, but the question of how to ascertain a `global state` of the system – the state of the collection of processes – is much harder to address.
The first problem is the absence of global time. If all processes had perfectly synchronised clocks then we could agree on a time at which each process would save its state – the result would be an actual global state of the system. But we know from CDK Chapter 10 that we cannot achieve perfect clock synchronisation. Given this constraint, the second problem is whether we can assemble a meaningful global state from local states recorded at different real times.

Mathematically, we can take any set of states of the individual processes to form a global state:

\[ S = (s_1, s_2, \ldots, s_n) \]

A global state corresponds to initial prefixes of the individual process histories. A cut of the system’s execution is a subset of its global history which is such a union of initial prefixes:

\[ C = h_1^\xi \cup h_2^\eta \cup \ldots \cup h_n^\omega \]

In general, the state \( s_i \) in the global state \( S \) corresponding to the cut \( C \) is that of \( p_i \) immediately after the last event processed by \( p_i \) in the cut – \( e_i^e \) \((i = 1, \ldots, n)\). The set of events \{ \( e_i^e \): \( i = 1, \ldots, n \) \} is called the frontier of the cut.

\[
\begin{array}{cccccc}
  & \mathbf{p_1} & \mathbf{p_2} & \mathbf{p_3} & \cdots & \mathbf{p_n} \\
 0 & s_0^1 & s_0^2 & s_0^3 & \cdots & s_0^n \\
 1 & s_1^1 & s_1^2 & s_1^3 & \cdots & s_1^n \\
 2 & s_2^1 & s_2^2 & s_2^3 & \cdots & s_2^n \\
 3 & s_3^1 & s_3^2 & s_3^3 & \cdots & s_3^n \\
 \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
 0 & s_0 & s_0 & s_0 & \cdots & s_0 \\
 \end{array}
\]

\text{Figure 1. A cut.}

Figure 1 shows a cut in which no event has occurred in the history of \( p_1 \), one event has occurred to \( p_2 \), no events have occurred at \( p_3 \) and two events at \( p_n \). It may be helpful to think of each of the processes as playing successive frames of a video tape. A cut results from slicing each of the \( n \) video tapes at some exact frame boundary.

What is the meaning of the state \( S \)? Suppose that \( s_i \) is \( p_i \)'s original state, and the remainder are the final states of the other processes. If \( p_2, \ldots, p_n \) first waited for \( p_1 \) to calculate and then send them an initial value, before updating their state, then this combination of local states cannot be a global state that ever existed.

The problem with the example is that the global state is inconsistent: the states of \( p_2, \ldots, p_n \) depend upon reception of a message which \( p_1 \)'s state shows it not to have sent yet! (see Figure 2).

Intuitively, if a global state reflects an event \( e \), then it should reflect all events that happened-before \( e \). We say that a cut \( C \) is a consistent cut if:

\[ e \in C \land (f \rightarrow e) \Rightarrow f \in C. \]

A consistent global state is one which corresponds to a consistent cut. We see now that not all runs pass through consistent global states, but all linearisations pass only through consistent global
states. We say that a state \( S' \) is \textit{reachable} from a state \( S \) if there is a linearisation that passes through \( S \) and then \( S' \).

We may characterise the execution of a distributed system as a series of transitions between global states of the system:

\[
S_0, e_1, S_1, e_2, S_2, \ldots
\]

In each transition, precisely one event occurs, at some single process in the system. This event is either the sending of a message, the receipt of a message, or an internal event. If two events happened simultaneously, we may nonetheless deem them to have occurred in a definite order – say ordered according to process identifiers. (Events that occur simultaneously must be concurrent: neither \textit{happened-before} the other.) A system evolves in this way through consistent global states (that is, in a linearisation). We may alter the ordering of concurrent events within a linearisation, and end up still with an execution that only passes through consistent global states. For example, if two successive events in a linearisation are the receipt by two processes of messages (over two different channels), then we may swap the order of these two events.

**Channel state**

Note that we have ignored the state of channels in this discussion. At any time, zero or more messages are in transit along a given channel – messages that have been sent by one process but which the process at the other end has not yet received. However, if we take the state of each process to record the set of messages it has placed on or taken delivery of from each connected

\[\square \quad \square \quad \begin{array}{l}
\square \\
\square \\
\square \\
\square \\
\end{array} \quad \square \quad \square \quad \square \quad \text{is inconsistent.}\]
channel, then we may derive the state of each channel from the states of the processes at either end.

In Section 3 we shall go on to examine algorithms for determining consistent global states. First, we continue this section on general features of distributed system models by considering the notions of time and failure.

2.4 Synchronous and asynchronous systems

We defined a system’s execution (a linearisation) solely as a sequence of events, and did not specify when each event occurred. If two processes executed exactly the same series of events, but one did so twice as fast as the other, then we would nonetheless say that they had the same history. One reason for this is to be found in the analysis of CDK Chapter 10: even if we did associate real time stamps with events, the time stamps in the histories of two different processes would be derived from clocks whose synchronisation is only approximate, so it would not always be possible to compare them meaningfully.

This reasoning motivates us to formulate a type of distributed system model from which we remove time altogether. We follow Hadzilacos and Toueg [1994] in defining an asynchronous system to be one in which there are no bounds on:

- process execution speeds – a process may take an arbitrarily long time between steps;
- message transmission delays – a message may be received an arbitrarily long time after it was sent;
- clock drift rates.

In other words, to assume that a system is asynchronous is to make no assumptions about the time intervals involved in any execution. A synchronous distributed system, conversely, is one in which upper and lower bounds exist on these quantities:

- each process takes a bounded time between steps;
- each message is transmitted over a channel and received in a bounded time;
- process’s local clocks may drift from real time only by a bounded rate.

Note that the last constraint only bounds the rate of local clock drift, not accuracy. The inclusion of the constraint allows us to use timeouts in protocols, for example to detect the failure of another process. If the communication channels are reliable, and if I receive no reply from a process in a maximum time measured on my local clock, then I may conclude in a synchronous system that the process has failed (assuming, of course, that I know the process is programmed to reply to my message).

Obviously, any real distributed system can be considered to be synchronous. Messages take at least one picosecond (10\(^{-12}\) s) and not more than a year to be delivered. Inter-step intervals are somewhere between a picosecond and a century. Left to themselves, working clocks don’t drift by more than a year every second.

We have chosen far-fetched values deliberately. The reader should consider what they think would be more realistic values. In what type of distributed system can the chosen values be guaranteed? Unless the values can be guaranteed, any design we make on the assumption of the chosen values will not be reliable. Herein lies the importance of the concept of an asynchronous system. If we can prove a system property on the asynchronous assumption, then that property must hold in any real distributed system.

Synchronous distributed systems can be built. What is required is for processes to be given guaranteed processor cycles (guaranteed by an operating system with real-time scheduling), for the networks and intermediate gateways also to provide real-time guarantees, and for processes to be

\[\text{---}\]

\(^2\)That is, assuming that the channel is reliable and never loses messages.

© Tim Kindberg
supplied with clocks with reasonable behaviour. Network capacity can be reserved and allocated so that the processes in the system receive guarantees of reasonably small message delays. Clearly, this is not a description of a general-purpose distributed system, which is subject to unpredictable loads.

Clock synchronisation

The algorithms in CDK Chapter 10 for synchronising clocks relied implicitly upon the system being synchronous. If, for instance, there are no bounds on message delays, then although we may use Cristian’s algorithm, for example, to try to synchronise two clocks, we can place no bound on the drift between them. The clocks may drift from one another by an hour during one execution, and by just ten milliseconds in another. There is nothing we can do to constrain this variation. Conversely, the guarantees provided in a synchronous system allow us to synchronise clocks.

2.5 Failure

Both processes and communication channels may fail in our distributed system. We must define the ways in which failure may occur if we are to understand its effect on our system. CDK Chapter 15, Section 15.3 gives Cristian’s classification of failures. Cristian’s taxonomy is phrased in terms of the failure of a service, but Hadzilacos and Toueg [1994] provide a related taxonomy that distinguishes between the failures of general processes and communication channels. Some of the failure modes are independent of time, and are of two main types: omission failures and arbitrary failures:

<table>
<thead>
<tr>
<th>Failure</th>
<th>Affects</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Omission:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Failstop</strong></td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may detect this state.</td>
</tr>
<tr>
<td><strong>Crash</strong></td>
<td>Process</td>
<td>Process halts and remains halted. Other processes may not be able to detect this state.</td>
</tr>
<tr>
<td><strong>Send-omission</strong></td>
<td>Process</td>
<td>Process generates a message, but the message is not placed on the channel.</td>
</tr>
<tr>
<td><strong>Omission</strong></td>
<td>Channel</td>
<td>A message inserted in the outgoing channel buffer never arrives at the other end’s buffer.</td>
</tr>
<tr>
<td><strong>Receive-omission</strong></td>
<td>Process</td>
<td>Message is delivered to the process’s input buffer, but the process never receives it.</td>
</tr>
<tr>
<td><strong>Arbitrary</strong></td>
<td>Process or channel</td>
<td>Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.</td>
</tr>
</tbody>
</table>

Omission and arbitrary failures may occur in both asynchronous and synchronous systems. But there are also time-related failures, relevant to synchronous systems only. In particular, these are failures that break one of the guarantees supposed to be provided by synchronous systems. Hadzilacos and Toueg provide the following time-related failure definitions relevant to synchronous systems:
3. Observing global states

Within this framework, we categorise failures according to their severity. *Benign* failures are all those other than arbitrary failures. That is, benign failures are failures of omission, clock failures and performance failures. Most failures in distributed systems are benign. An algorithm for dealing with arbitrary failures is studied in CDK Section 15.3.

### 3.1 Motivation

Actually, we are usually not so much interested in knowing all the details of the state of the system as we are in the question of whether some predicate was or may have been true at some point in the execution. This is a problem with a practical motivation and we outline the examples of distributed garbage collection, deadlock detection, termination detection and debugging.

#### Distributed garbage collection

An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system (Figure 3). The memory taken up by that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there no references to it either in a process or in a message en route to a process.

#### Distributed deadlock detection

CDK Section 14.5 considers this problem, which is a distributed version of the well-known pitfall of a cycle in the graph of wait-for dependency between processes (Figure 4). No process in the cycle may make any progress.
**Distributed termination detection.** The problem here is to detect that a distributed algorithm has terminated. Detecting termination is a problem which sounds deceptively easy to solve: do we not need simply to test whether each process has halted? To see that this is not so, consider a distributed algorithm executed by two processes \( p \) and \( q \), each of which may request values from the other. Instantaneously, we may find that a process is either active or is passive – a passive process is not engaged in any activity of its own, but is prepared to respond with a value requested by the other. Suppose we discover that \( p \) is passive, and then that \( q \) is passive. May we conclude that the algorithm has terminated? To see that the answer is no, consider the following scenario: when we tested \( p \) for passivity, a message was on its way from \( q \), which became passive immediately after sending it (Figure 5). On receipt of the message, \( p \) became active again – after we found it to be passive. The algorithm had not terminated.

![Figure 5. Detecting termination.](image)

The phenomena of termination and deadlock may seem similar, but they are different problems. First, a deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated. Second, process passivity is not the same as waiting in a deadlock cycle: a deadlocked process is attempting to perform a further action, for which another process waits; a passive process is not engaged in any activity.

**Distributed debugging and monitoring.** Smith has written an application in which a single token is passed between processes to enforce synchronisation. Unfortunately, there is a bug in the program, and she suspects that under certain circumstances there are two tokens in the system, breaking her consistency constraints. How may she detect whether two processes held tokens at the same time?

Detecting the different conditions that we have just described amounts to evaluating a global state predicate. A global state predicate is a function that maps from the set of global states of processes \( p_i, i = 1, \ldots, n \) in the system to \{True, False\}. One of the useful characteristics of the predicates associated with the state of an object being garbage, of the system being deadlocked or the system being terminated is that they are all stable: once the system enters a state in which the predicate is True, it remains True in all future states reachable from that state. By contrast, when we monitor or debug an application we are often interested in non-stable predicates, such as in the token example. Even if the application reaches a state in which two tokens exist, it need not stay in that state.

### 3.2 Safety and liveness

It is worth noting here two concepts of relevance to global state predicates: safety and liveness. Suppose there is an undesirable property \( \alpha \) that is a predicate of the system’s global state – for example, \( \alpha \) could be the property of being deadlocked. Let \( S_0 \) be the original state of the system. Safety with respect to \( \alpha \) is the assertion that \( \alpha(S) \) is False for all states \( S \) reachable from \( S_0 \). Conversely, let \( \beta \) be a desirable property of a system’s global state – for example, the property of reaching termination. Liveness with respect to \( \beta \) is the property that \( \beta(S) \) is True for some state \( S \) reachable from \( S_0 \) in any given linearisation (consistent run).
3.3 The snapshot algorithm of Chandy & Lamport

Chandy and Lamport [1985] describe a “snapshot” algorithm for determining global states of distributed systems, which we now describe. The goal of the algorithm is:

- to record a set of process and channel states such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

We shall see that the state that the snapshot algorithm records has convenient properties for evaluating stable global predicates.

The algorithm records state locally at processes: it does not give a method for gathering the global state at one site. An obvious method for gathering the state is for all processes to send the state they recorded to a designated collector process, but we shall not address this issue further here.

The algorithm assumes:

- neither channels nor processes fail – every message sent is eventually received intact;
- channels provide FIFO message delivery;
- the graph of processes and channels is strongly connected (there is a path between any two processes);
- any process may initiate a global snapshot at any time;
- the processes continue their execution and send and receive normal messages while the snapshot takes place.

For each process \( p \) let the incoming channels be those incident at \( p \) over which other processes send it messages; similarly, \( p \)’s outgoing channels are those on which it sends messages to other processes (Figure 6). The essential idea of the algorithm is as follows. Each process records its state and also for each incoming channel the set of messages sent to it between the moment of recording its state and the point at which the sender recorded its state. This arrangement allows us to record the states of processes at different times, but to account for the differentials between process states in terms of messages transmitted but not yet received.

The algorithm proceeds through use of special marker messages, which are distinct from any other messages the processes send, and which may be sent and received while the processes proceed with their normal execution. The algorithm is as follows:

**Marker receiving rule for process \( p \).**

On \( p \)’s receipt of a marker message over channel \( c \):

- If \( p \) has not yet recorded its state, it:
  - records its process state now;
  - records the state of \( c \) as the empty set;
  - turns on recording of messages arriving over other incoming channels;
else

\( p \) records the state of \( c \) as the set of messages it has received over \( c \) since it first received a marker message.

end

**Marker sending rule for process \( p \).**

For each outgoing channel \( c \):

\( p \) sends a marker message over \( c \) after it has recorded its state, and before it sends any other message over \( c \).

Any process may begin the algorithm at any time, by recording its state and beginning to record messages arriving over incoming channels. In fact, several processes may initiate recording concurrently in this way.

We illustrate the algorithm for a system of two nondeterministic processes, \( p \) and \( q \) connected by two unidirectional channels, \( c \) and \( c' \). The processes have state transition diagrams shown in Figure 7 (taken from [Chandy and Lamport 1985]). Each process may spontaneously emit a message (\( X \) and \( X' \) respectively), causing it to change its internal state. Similarly, receipt of the other process’s message causes a state transition.

Figure 8 shows an execution of the system while the state is recorded. Process \( p \) records its state in the global state \( S_0 \), when \( p \)’s state is A. Process \( p \) then emits a marker message followed by a message \( X \) over channel \( c \). The system enters global state \( S_1 \). Before \( q \) receives the marker, it emits a message \( X' \) (giving state \( S_2 \)) which \( p \) receives. Process \( q \) then receives the marker, records its state and sends a marker message over \( c' \). The final global state is \( S_3 \). Process \( q \) records the state of channel \( c \) as being the empty sequence; process \( p \) records the state of channel \( c' \) as being the single message \( X' \) (which it received after it recorded its state and before it received \( q \)’s marker message). The final recorded states are: \( p: A; q: D; c: \{\}; c': \{X'\} \). Note that this state differs from all the global states through which the system actually passed.

**Termination**

We assume that a process that has received a marker message (1) records its state within a finite time, and (b) sends marker messages over each outgoing channel within a finite time (even when it no longer needs to send application messages over these channels). If there is a path of
communication channels and processes from a process $p$ to a process $q$, then it is clear on these assumptions that $q$ will record its state a finite time after $p$ recorded its state. Since we are assuming the graph of processes and channels to be strongly connected, it follows that all processes will have recorded their states and the states of incoming channels a finite time after some process initially records its state.

**Characterising the observed state**

The state recorded by this algorithm is consistent. Let $e$ and $f$ be events occurring at $p$ and $q$ respectively, and let $f \rightarrow e$. If $e$ occurred before $p$ recorded its state, then $f$ must have occurred before $q$ recorded its state. Otherwise, consider a sequence of messages $msg_1, ..., msg_m$ giving rise to the relation $f \rightarrow e$ (see Figure 9). By FIFO ordering over the channels these messages traverse, and by the marker sending rule, a marker message would have reached $p$ before the event $e$, contradicting our assumption.

We may further establish a reachability relation between the observed global state and the initial and final global states when the algorithm runs. Let $sys = e_0, e_1, ...$ be the system as it executed (where two events occurred at exactly the same time, we order them according to process identifiers). Let $S_i$ be the global state just before the first process recorded its state, let $S_φ$ be the state in which the snapshot algorithm terminates, and let $S^*$ be the recorded state. We shall find a permutation of $sys$, $sys' = e'_0, e'_1, ...$ such that all three states $S_i$, $S_φ$, and $S^*$ occur in $sys'$, and $S'$ is reachable from $S_i$, and $S_φ$ is reachable from $S^*$ in $sys'$ (see Figure 10, in which the upper run corresponds to $sys$, and the lower to $sys'$). In other words:
(1) \( \forall i : i < t \lor i \geq \phi \Rightarrow e_i' = e_i \)

(2) the subsequence \((e_i', t \leq i < \phi)\) is a permutation of \((e_i, t \leq i < \phi)\)

(3) \( \forall i : i < t \lor i \geq \phi \Rightarrow S_i' = S_i \)

(4) \( \exists k : t \leq k \leq \phi, S^* = S_k \)

Figure 10. Reachability between states \(S_i\), \(S^*\) and \(S_{\phi}\).

We derive \(sys'\) from \(sys\) by first splitting all events in \(sys\) into \textit{prerecording} events and \textit{postrecording} events. A prerecording event is one that occurred before the process at which it occurred recorded its state; all other events are postrecording events. A postrecording event may occur before a prerecording event in \(sys\), if the events occur at different processes. (Of course no postrecording event may occur before a prerecording event at the same process.)

We shall show how we may order all prerecording events before postrecording events. Suppose that \(e_j\) is a postrecording event, and \(e_{j+1}\) is a prerecording event. It cannot be that \(e_j \rightarrow e_{j+1}\). For then these two events would be the sending and receiving of a message, respectively. A marker message would have to have preceded the message, making the reception of the message a postrecording event. We may therefore swap the two events without violating the \textit{happened-before} relation (that is, the resultant run remains a linearisation), and without introducing new process states.

We continue in this way as necessary until we have ordered all prerecording events prior to all postrecording events, with \(sys'\) the resulting execution, and \(S_k\) the state just before the first postrecording event. Clearly, the process states recorded in \(S_k\) are the same as the process states recorded in \(S^*\). Also, the set of messages in each channel in \(S_k\) is the same as for \(S^*\), because each process records the messages sent over an incoming channel after it has recorded its own state and before it receives a marker. These messages are the same as the set of prerecorded sends over the channel, less any received before the state is recorded.

**Snapshots and stable states**

We have established the reachability between \(S_i\), \(S^*\) and \(S_{\phi}\) because the reachability property is useful for detecting stable predicates. In general, any non-stable predicate we establish as being \textit{True} in the state \(S^*\) may or may not have been \textit{True} in the actual execution whose global state we recorded. However, if a stable predicate is \textit{True} in the state \(S^*\) then we may conclude that the predicate is \textit{True} in the state \(S_{\phi}\), since by definition a stable predicate that is \textit{True} of a state \(S\) is also \textit{True} of any state reachable from \(S\). Similarly, if the predicate evaluates to \textit{False} for \(S^*\), then it must be \textit{False} for \(S_i\) also.
### 3.4 A centralised algorithm for distributed debugging

We now examine an algorithm for recording a system’s global state so that we may make useful statements about whether a transitory state – as opposed to a stable state – may have occurred or definitely occurred in an actual execution. Chandy and Lamport’s snapshot algorithm collects state in a distributed fashion, and we pointed out how the processes in the system could send the state they gather to a monitor process for collection. The algorithm we shall describe is centralised: system processes send their state to a process called a monitor, which assembles a globally consistent state from what it receives [Marzullo & Neiger 1991]. We consider the monitor to lie outside the system we observe, examining the operation of a distributed application.

Our aim is to determine cases where a given global state predicate $\phi$ was definitely True at some point in the execution we observed, and cases where it was possibly True. For example, in a distributed system controlling a factory we may be interested in whether all the valves (controlled by different processes) were open at some time. Another example, which we have already mentioned, is the question of whether the number of tokens in a system supposed to maintain mutual exclusion is ever greater than one. Of course, we are interested in knowing whether these states of affairs actually occurred in an execution. But we may also be interested to know whether they may have occurred.

The notion ‘possibly’ arises as a natural concept because we may extract a consistent global state $S'$ from an executing system and find that $\phi(S')$ is True. But no single observation of a consistent global state allows us to conclude whether a non-stable predicate ever evaluated to True in the actual execution.

The notion ‘definitely’ does apply to the actual execution, and not to a run that we have extrapolated from it. It may sound paradoxical for us to consider what happened in an actual execution. However, it is possible to evaluate definitely $\phi$ by considering all linearisations of the observed events. We now define the notions of possibly $\phi$ and definitely $\phi$ for a predicate $\phi$ in terms of linearisations of $H$, the history of the system’s execution.

**Possibly $\phi$.** The statement possibly $\phi$ means that there is a consistent global state $S = (s_1, s_2, ..., s_n)$ through which a linearisation of $H$ passes such that $\phi(S)$ is True.

**Definitely $\phi$.** The statement definitely $\phi$ means that for all linearisations $L$ of $H$, there is a (consistent) global state $S_L = (s^L_1, s^L_2, ..., s^L_n)$ through which $L$ passes such that $\phi(S_L)$ is True.

When we use Chandy and Lamport’s snapshot algorithm we may assert possibly $\phi$ if $\phi(S')$ happens to be True. But in general evaluating possibly $\phi$ entails a search through all consistent global states derived from the observed execution. Only if $\phi$ evaluates to False for all consistent global states is it not the case that possibly $\phi$. Note also that while we may conclude definitely ($\neg \phi$) from $\neg$possibly $\phi$, we may not conclude $\neg$possibly $\phi$ from definitely ($\neg \phi$). The latter is the assertion that $\neg \phi$ holds at some state on every linearisation: $\phi$ may hold for other states.

We now describe:

- how the process states are collected
- how the monitor extracts consistent global states
- how the monitor evaluates possibly $\phi$ and definitely $\phi$, in both asynchronous and synchronous systems.
Collecting the state

The system processes send their initial state to the monitor process initially, and thereafter from time to time. The monitor process records the messages from each process $p_i$ in a queue $Q_i$ (see Figure 11). This activity may delay their normal execution but does not interfere with it. Clearly, there is no need to send the state except initially and when it changes. There are two optimisations to reduce the message traffic to the monitor. First, the global state predicate may depend only on certain parts of the processes’ states. For example, it may depend only on the states of particular variables. So the system processes need only send the relevant state to the monitor process. Second, they need only send their state at times when the predicate may become True or cease to be True. There is no point in sending changes to the state that do not affect the predicate’s value. For example, consider a system with two processes. Process $p_1$ has local variables $x$ and $y$, process $p_2$ has local variables $u$ and $v$. Let the predicate we are interested in be $x = y = u = v$. The processes need only notify the monitor when the values of their own two variables become equal or cease to be equal. When they send their state, they supply only the values of the two variables concerned.

Finding consistent global states

The monitor must assemble only consistent global states against which it evaluates $\phi$. Recall that a global state corresponding to a cut $C$ is consistent if and only if for any events $e$ and $f$:

$$e \in C \land (f \rightarrow e) \Rightarrow f \in C.$$ 

So the processes must send their states to the monitor with sufficient information to allow it to detect whether any two process states result directly from concurrent events, or events related by the happened-before relation ‘$\rightarrow$’.

In CDK Chapter 10 we introduced the notions of logical time and logical clocks. If $C(e)$ is the logical timestamp of an event $e$, then:

$$e \rightarrow f \Rightarrow C(e) < C(f).$$

However, the converse is not true: from the logical timestamps of two events we can deduce nothing about whether one happened-before the other.

© Tim Kindberg
Therefore it would be insufficient for system processes to append the values of their logical clocks when they send their state to the monitor process. Fortunately, there is a development of logical clocks called vector clocks, which do allow us to test for the happened-before relationship. In other words, if $V(e)$ is the vector timestamp of event $e$, then:

$$e \rightarrow f \iff V(e) < V(f).$$

Vector clocks are introduced in CDK Chapter 11, but we now give a brief overview.

**Vector Clocks**

A vector clock used in a system of $n$ processes is an array of $n$ integers. Each process keeps its own vector clock, which it uses to timestamp local events. So in a system of $n$ processes there are $n$ vector clocks $V_i, i = 1, 2, \ldots, n$. Like logical timestamps, processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks as follows:

- **VC1:** Initially, $V_i[j] = 0$, for $i, j = 1, 2, \ldots, n$.
- **VC2:** Just before $p_i$ timestamps an event, $V_i[i] := V_i[i] + 1$.
- **VC3:** $p_i$ includes the value of $V_i$ in every message it sends.
- **VC4:** When $p_i$ receives a timestamp $M$ in a message, it sets $V_j[j] := \max(V_j[j], M[j])$, for $j = 1, 2, \ldots, n$.

For a vector clock $V_i$, $V_i[i]$ is the number of events that $p_i$ has timestamped, and $V_i[j]$ $(j \neq i)$ is the number of events that have occurred at $p_j$ that $p_i$ currently knows about. (Process $p_j$ may have timestamped more events by this point, but no information has flowed to $p_i$ about them yet.)

We may compare vector timestamps as follows:

$$V_1 \leq V_2 \iff V_1[i] \leq V_2[i], \text{ for } i = 1, 2, \ldots, n.$$

$$V_1 < V_2 \iff V_1 \leq V_2 \land V_1 \neq V_2.$$

For the purposes of evaluating global state predicates, it is not necessary for the processes to timestamp every event that occurs. They need only timestamp:

- internal events that may affect the value of the predicate (corresponding to the state values that they send to the monitor process), and
- communication events – when they communicate with one another.

While the system processes enclose timestamps in their messages to the monitor process, it is not necessary for them to update their timestamps at those points. The monitor is not involved in the application, and never sends messages to the system processes.

Figure 12 (taken from [Marzullo and Neiger 1991]) shows the execution of three processes, each of which can switch on and off its own light. Event $a$, for example, is process $p_1$ switching on its light, event $f$ is process $p_3$ switching off its light. The figure shows all the vector timestamps involved in this execution.

**Vector clocks and consistent states**

We now require that the system processes enclose their vector clock values with their state messages sent to the monitor process. We have already said that the monitor process keeps the state messages arriving from process $p_i$ in a queue $Q_i$. Each queue is ordered in sending order, which can immediately be established by examining the vector timestamps. Of course, the monitor
process may deduce nothing about the relationship between states sent by different processes from their arrival order. It must instead examine the vector timestamps of the state messages.

Let \( S = (s_1, s_2, \ldots, s_n) \) be a global state drawn from the state messages that the monitor process has received. Let \( V(s_i) \) be the vector timestamp of state \( s_i \) received from \( p_i \). Then \( S \) is a consistent global state if and only if:

\[
V(s_i)[i] \geq V(s_j)[i], \text{ for } i, j = 1, \ldots, n.
\]

(condition CGS\(^3\))

This says that the number of \( p_i \)'s events known at \( p_j \) when it sent \( s_j \) is no more than the number of events that had occurred at \( p_i \) when it sent \( s_i \). In other words, if one process’s state depends upon another (according to happened-before ordering), then the global state also encompasses the state upon which it depends.

In summary, we now possess a method whereby the monitor process may establish whether a given global state is consistent, using the vector timestamps kept by the system processes and piggy-backed with the state messages they send to it.

**Evaluating possibly and definitely \( \phi \)**

Figure 13 (taken from [Marzullo & Neiger 1991]) shows the lattice of consistent global states corresponding to the execution of the two processes on the right. This structure captures the relation of reachability between consistent global states. The nodes denote global states and the edges denote possible transitions between these states. The global state \( S_{00} \) has both processes in their initial state; \( S_{01} \) has \( p_1 \) still in its initial state and \( p_2 \) in the next state in its local history. The state \( S_{20} \) is not consistent, because of the first message sent from \( p_2 \) to \( p_1 \), so it does not appear in the lattice.

The lattice is arranged in levels with, for example, \( S_{00} \) in level 0, \( S_{01} \) in level 1. In general, \( S_{xy} \) is in level \( x + y \). A consistent run traverses the lattice from any global state to any global state reachable from it in steps that each move to a state on the next level – that is, in each step some process experiences one event. For example, \( S_{56} \) is reachable from \( S_{34} \), but \( S_{56} \) is not reachable from \( S_{64} \).

The lattice shows us all the linearisations corresponding to a history. It is now clear in principle how a monitor process should evaluate possibly \( \phi \) and definitely \( \phi \). To evaluate possibly \( \phi \), the

\(^3\)See appendix for proof.
monitor process starts at the initial state, and steps through all consistent states reachable from that point, evaluating $\phi$ at each stage. It stops when $\phi$ evaluates to True. To evaluate definitely $\phi$, the monitor process must attempt to find a set of states through which all linearisations must pass, and at each of which $\phi$ evaluates to True. For example, if $\phi(S_{43})$ and $\phi(S_{34})$ are both True then, since all linearisations pass through them, definitely $\phi$ holds.

**Possibly $\phi$**

To evaluate possibly $\phi$ the monitor process must traverse the lattice of reachable states, starting from the initial state $(s_1^0,..,s_n^0)$. The algorithm is:

```
Level := 0; States := { (s_1^0,..,s_n^0) };

while ( for all S \in States  \phi(S) = False )
    Level := Level+1;
    States := { S' : level(S') = Level \wedge S' reachable from some S \in States };

output "possibly $\phi$";
```

The algorithm assumes that the execution is infinite. It may easily be adapted for a finite execution. The monitor process may discover the set of consistent states in level Level+1 reachable from a given consistent state in level Level by the following simple means. Let $S = (s_1,..,s_n)$ be a consistent state. Then a consistent state in the next level reachable from $S$ is of the form
Figure 14. Illustration of the algorithm for definitely $\phi$. ‘T’ means that $\phi$ is True at the state; ‘F’ means it is False.

$S' = (s'_1, \ldots, s'_n)$, which differs from $S$ only by containing the next state of some process $p_i$. The state $S'$ is reachable from $S$ only if:

$$\text{for } j = 1, \ldots, n; j \neq i: V(s_j)[j] \geq V(s'_j)[j].$$

This condition comes from considering condition CGS above, and from the fact that $S$ was already a consistent global state. A given state may in general be reached from several states at the previous level, so the monitor process takes care to evaluate the consistency of each state only once.

**Definitely $\phi$**

To evaluate definitely $\phi$ the monitor process again traverses the lattice of reachable states a level at a time, starting from the initial state $(s'_1^0, \ldots, s'_n^0)$. The algorithm, which again assumes that the execution is infinite, is:

```
Level := 0;
if $\phi(s'_1^0, \ldots, s'_n^0) = True$ then Remain := $\emptyset$ else Remain := { $(s'_1^0, \ldots, s'_n^0)$ } fi;

while (Remain $\neq \emptyset$)
    Level := Level + 1;
    Remain := $\{S': \text{level}(S') = Level \land S'$ reachable from some $S \in$ Remain$\}$;
    Remain := $\{S \in$ Remain:$\phi(S) = False$};

élihW
output “definitely $\phi$”;  
```

This algorithm maintains the set $\text{Remain}$. This is the set of states at the current level that may be reached on a linearisation from the initial state, but without traversing any state for which $\phi$ evaluates to $True$. As long as such a linearisation exists, we may not assert definitely $\phi$: the execution could have taken this linearisation and $\phi$ would be $False$ at every stage along it. Once we reach a level for which no such linearisation exists, we may conclude definitely $\phi$. In Figure 14, at

---

4Marzullo & Neiger [1991] include a condition for $j = 1, \ldots, n; j \neq i$: $V(s'_j)[i] \geq V(s_j)[i]$. This is redundant given the consistency of $S$, for $V(s'_j)[i] > V(s_j)[i]$, and $V(s_j)[i] \geq V(s'_j)[i]$. 

© Tim Kindberg
level three the set \textit{Remain} consists of only one state, which is reachable by a linearisation on which all states are \textit{False} (marked in bold lines). Note that there is another linearisation reaching this state, which passes through a state with \textit{\phi True}. The only states to consider at level four are those marked ‘?’. If \textit{\phi} evaluates to \textit{True} for both states, then we may conclude \textit{definitely} \textit{\phi}. Otherwise, the algorithm must continue beyond level four.

\textbf{Cost}

The algorithms we have just described are combinatorically explosive. Suppose that \textit{k} is the maximum number of events at a single process. Then the algorithms we have described entail \textit{O}(\textit{k}^\textit{n}) comparisons (the monitor process compares the states of each of the \textit{n} system processes with one another).

There is also a space cost to these algorithms of \textit{O}(\textit{k} \times \textit{n}). However, we observe that the monitor process may delete an item \textit{s}_i from a queue \textit{Q}_i when no other item of state arriving from another system process could possibly be involved in a consistent global state with \textit{s}_i. That is, we must have:

\[ V(s_j^{last})[i] > V(s_i)[i], \]

where \textit{s}_j^{last} is the last state that the monitor process has received from process \textit{p}_j.

We now examine how, in the special case of synchronous systems, we can eliminate some comparisons from the monitor process’s calculations by using synchronised clocks.

\textbf{Evaluating possibly and definitely \textit{\phi} in synchronous systems}

The algorithms we have given so far work in an asynchronous system: we have made no timing assumptions. But the price paid for this is that the monitor examines consistent global states \textit{(s}_1,\ldots,\textit{s}_n) for which any two local states \textit{s}_i and \textit{s}_j may have occurred an arbitrarily long time apart in the actual execution of the system. Our requirement, by contrast, is to consider only those global states that the actual execution could in principle have traversed.

In a synchronous system, suppose that the processes keep their physical clocks synchronised within known bounds, and that the system processes provide physical timestamps as well as vector timestamps with their state messages. Then the monitor process need consider only those consistent global states whose local states could possibly have existed simultaneously, given the approximate synchronisation of the clocks. With good enough clock synchronisation, this will be many fewer than the set of all globally consistent states.

We now give an algorithm to exploit synchronised clocks in this way. We assume that each process \textit{p}_i (\textit{i} \geq 1) and the monitor process, which we shall call \textit{p}_0, keeps a physical clock \textit{C}_i. These are synchronised to within a known bound \textit{\epsilon}; that is, at the same real time:

\[ |C_i - C_j| \leq \epsilon, \quad i, j = 0, \ldots, n. \]

The system processes send both their vector time and physical time with their state messages to the monitor process. The monitor process now applies a condition that not only tests for consistency of a global state \textit{S} = (\textit{s}_1, \textit{s}_2, \ldots, \textit{s}_n), but also tests whether each pair of states could have happened at the same real time, given the physical clock values. In other words, for \textit{i}, \textit{j} = 1, \ldots, \textit{n}:

\[ V(s_j)[i] \geq V(s_i)[i], \]

\textit{s}_i and \textit{s}_j could have occurred at the same real time.

© Tim Kindberg
Figure 15. Overlapping intervals between state transitions. The shaded projections represent the differences of the two processes’ clocks.

The first clause is taken from the condition CGS which we used earlier. For the second clause, note that $p_i$ is in the state $s_i$ from the time it first notifies the monitor process, $C_i(s_i)$, to some later local time $L(s_i)$, say, when the next state transition occurs (see Figure 15). For $s_i$ and $s_j$ to have obtained at the same real time we thus have, allowing for the bound on clock synchronisation:

$$(C_i(s_i) - \varepsilon \leq C_j(s_j) \leq L(s_i) + \varepsilon) \lor (C_j(s_j) - \varepsilon \leq C_i(s_i) \leq L(s_j) + \varepsilon).$$

The first clause is taken from the condition CGS which we used earlier. For the second clause, note that $p_i$ is in the state $s_i$ from the time it first notifies the monitor process, $C_i(s_i)$, to some later local time $L(s_i)$, say, when the next state transition occurs (see Figure 15). For $s_i$ and $s_j$ to have obtained at the same real time we thus have, allowing for the bound on clock synchronisation:

$$(C_i(s_i) - \varepsilon \leq C_j(s_j) \leq L(s_i) + \varepsilon) \lor (C_j(s_j) - \varepsilon \leq C_i(s_i) \leq L(s_j) + \varepsilon).$$

If state of $p_i$ changed in this interval, monitor would know by now

Figure 16. Has the process’s state changed by the time the monitor examines it?

The monitor process must calculate a value for $L(s_i)$, which is measured against $p_i$’s clock. If the monitor process has received a state message for $p_i$’s next state $s'_i$, then $L(s_i)$ is $C_i(s'_i)$. Otherwise, the monitor process estimates $L(s_i)$ as $C_0 - \text{maxMsgTime}$, where $C_0$ is the monitor’s current local clock value, and $\text{maxMsgTime}$ is the maximum time between the sending and receipt of a state message (see Figure 16).
Summary
We began by giving an overview of distributed system models. We distinguished message passing from shared-memory models. We also distinguished synchronous from asynchronous system models. In a synchronous system we make assumptions about bounds on message delays, processing rates and clock drift rates. In an asynchronous system we make no such assumptions. The advantage of assuming asynchronicity is the general applicability of the results we obtain.

We introduced the concepts of events, local and global histories, cuts, local and global states, runs, consistent states, consistent runs (linearisations), and reachability. A consistent state or run is one that is in accord with the happened-before relation. This relation defines a partial order on events.

We went on to consider the problem of recording a consistent global state by observing a system’s execution. Our objective was to evaluate a predicate on this state. An important class of predicates are the stable predicates. We described the snapshot algorithm of Chandy and Lamport, which captures a consistent global state and allows us to make assertions about whether a stable predicate holds in the actual execution. We went on to give Marzullo and Neiger’s algorithm for deriving assertions about whether a non-stable predicate held or may have held in the actual run. The algorithm employs a monitor process to collect states. The monitor examines vector timestamps to extract consistent global states, and it constructs and examines the lattice of all consistent global states. This algorithm involves great computational complexity, but is valuable for understanding and may be of some practical benefit in real systems where relatively few events change the global predicate’s value. The algorithm has a more efficient variant in synchronous systems, where clocks may be synchronised.

We shall develop the failure models and shared memory models elsewhere.

Tim Kindberg, QMW

References (not in CDK)


Appendix: proof of condition CGS
Lemma
\[ e_i \rightarrow e_j \iff V(e_i)[i] \leq V(e_j)[i]. \]
The event \( e_i \) occurs at \( p_i \), and \( e_j \) occurs at \( p_j \) (we assume that \( i \neq j \), or the proof is trivial).

(1) Assume \( e_i \rightarrow e_j \). Then there is a chain of messages between \( e_i \) and \( e_j \). By rule VC4 for vector clocks, \( V(e_i) \leq V(e_j) \). Therefore \( V(e_i)[i] \leq V(e_j)[i] \).

(2) Assume \( V(e_i)[i] \leq V(e_j)[i] \). If \( \neg e_i \rightarrow e_j \), then there is no chain of messages between the occurrences of \( e_i \) and \( e_j \). But then, by VC2 and VC4, \( V(e_i)[i] > V(e_j)[i] \) and we have a contradiction. So \( e_i \rightarrow e_j \).

© Tim Kindberg
Proposition

The cut $C = (e_1^c, \ldots, e_n^c)$ corresponding to global state $S = (s_1, \ldots, s_n)$ is consistent iff $V(s_i)[i] \geq V(s_j)[i]$, for $i, j = 1, \ldots, n$.

(1) Assume that $V(s_i)[i] \geq V(s_j)[i]$, for $i, j = 1, \ldots, n$. We shall show that the cut is consistent by deriving a contradiction. Assume that $e_j \in C \land e_i \rightarrow e_j$ but that $e_i \notin C$. The situation is shown in the following figure, which only shows the events in the cut belonging to $p_i$ and $p_j$.

It is easy to see from the figure using rule VC2 and the above lemma that $V(e_i^c)[i] < V(e_i)[i]$, $V(e_i)[i] \leq V(e_j)[i]$ and $V(e_i)[i] \leq V(e_j^c)[i]$.

Therefore, $V(e_i)[i] < V(e_j)[i]$ and we have the necessary contradiction. So $C$ is consistent.

(2) Assume that $C$ is consistent. If there is no $e_i \in C$ such that $e_i \rightarrow e_j^c$, then by VC4 and considering the absence of a chain of messages between processes $p_i$ and $p_j$, $V(e_j^c)[i] = 0$. Therefore, trivially, $V(e_i^c)[i] \geq V(e_j^c)[i]$. Otherwise, choose the latest $e_i \in C$ such that $e_i \rightarrow e_j^c$ (see the following figure).

By considering the chain of messages between events $e_i$ and $e_j^c$ and applying VC4, $V(e_i)[i] = V(e_j^c)[i]$. We also have $V(e_i^c)[i] \geq V(e_i)[i]$ by consistency of $C$, so $V(e_i)[i] \geq V(e_j^c)[i]$ in this case also.

QED.