

Distributed Systems

June 27, 2004

Books

- *Distributed Computing* – H. Attiya and J. Welch – McGraw-Hill, 1998, ch. 1
- *Concurrent Systems* – J. Bacon – Addison-Wesley 1994, ch. 1
- *Distributed Systems* – Coulouris et al. – A-W 1993, ch. 1
- *Distributed Systems* – Mullender (ed.) – A-W 1993, ch. 1 and 2
- *Distributed Algorithms* – N. Lynch – Morgan Kaufman 1996, ch. 1

Introduction

A distributed system is a system composed of several computers that are connected in some way and that together provide some service or achieve some goal, e.g.

- communication, e.g. cellphones, email
- process control, e.g. in aircraft
- information processing, e.g. financial transactions
- scientific computing, e.g. weather forecasting

The hardware of a distributed system may for instance be a VLSI chip, a shared-memory multiprocessor machine, a LAN of workstations or PCs, or a global network.

In this course we study some ideas, concepts and techniques that are useful in understanding, designing and implementing distributed systems.

1 A Mathematical Model

We begin by describing a mathematical model that can be used to express many distributed systems.

Consider a finite connected graph (\mathbf{P}, \mathbf{L}) . We call the nodes *processes* and the edges *links*. A link makes possible communication of data from a set \mathbf{D} between the processes that it joins.

Example: Suppose the graph has 12 processes P_0, \dots, P_{11} connected in a ring, i.e. $\mathbf{P} = \{P_0, \dots, P_{11}\}$ and $\mathbf{L} = \{(P_i, P_{i+1}) | 0 \leq i < 12\}$ where arithmetic is mod 12. Let $\mathbf{D} = \mathbf{N} \cup \{\text{done}\}$.

[Diagram goes here]

The set \mathbf{M} of messages is $\mathbf{P} \times \mathbf{D} \times \mathbf{P}$.

Example: The message (P_3, done, P_4) is the datum *done* set from P_3 to P_4 .

The *state* of a process at a given moment is determined by the control state it is in, the data it is storing, and the arrived messages that have been delivered to it but not examined by it.

Example: Suppose each P_i has an associated name $n_i \in \mathbf{N}$. Let the set \mathbf{S} of states be $\mathbf{Q} \times \mathbf{N} \times \mathcal{F}(\mathbf{M})$ where $\mathbf{Q} = \{\text{init}, \text{active}, \text{leader}, \text{follower}\}$ and $\mathcal{F}(\mathbf{M})$ is the set of all finite multisets¹ of messages. A state of P_2 might be $(\text{active}, 20, \{(P_1, 0, P_2), (P_1, 10, P_2)\})$.

A *configuration* of a distributed system describes the state of each of its processes and the collection of messages that have been sent but not delivered.

Example: The set \mathbf{C} of configurations is $\mathbf{S}^{12} \times \mathcal{F}(\mathbf{M})$.

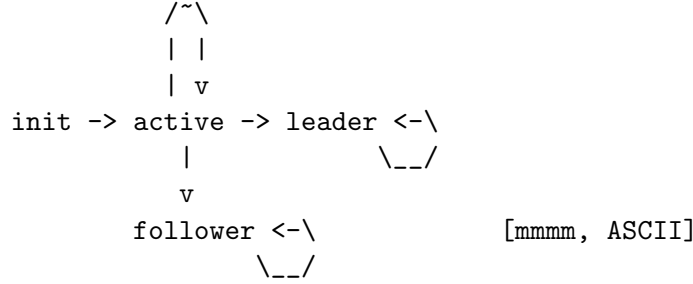
Each process has a *transition function* that determines what, if anything, it may do in a given state. Transition functions are sometimes described explicitly, as in the next example, and sometimes implicitly, by some text, pseudo-code, or program.

Example: The transition function of P_i , for $0 \leq i < 12$, is $\delta_i : \mathbf{S} \rightarrow \mathbf{S} \times \mathcal{F}(\mathbf{M})$ defined by:

$$\begin{aligned} \delta_i(\text{init}, n, \text{in}) &= ((\text{active}, n, \text{in}), \{(P_i, n, P_{i+1})\}) \\ \delta_i(q, n, \text{in}) &= ((q, n, \emptyset), \emptyset) \\ &\quad \text{if } q=\text{leader} \text{ or } q=\text{follower} \text{ or } (q=\text{active} \text{ and } \text{in}=\emptyset) \\ \delta_i(\text{active}, n, \text{in}) &= \\ &\quad \text{if } \text{done} \in \text{data}(\text{in}) \text{ then } ((\text{follower}, n, \emptyset), \{(P_i, \text{done}, P_{i+1})\}) \\ &\quad \text{else if } n \in \text{data}(\text{in}) \text{ then } ((\text{leader}, n, \emptyset), \{(P_i, \text{done}, P_{i+1})\}) \\ &\quad \text{else if } n > \max(\text{in}) \text{ then } ((\text{active}, n, \emptyset), \emptyset) \\ &\quad \text{else if } n < \max(\text{in}) \text{ then } ((\text{active}, n, \emptyset), \{(P_i, \max(\text{in}), P_{i+1})\}) \\ &\quad \text{where } \text{data}(\text{in}) \text{ is the set of data in the message set } \text{in}, \\ &\quad \text{and } \max(\text{in}) \text{ is the largest integer in that set (and } \text{in} \neq \emptyset) \end{aligned}$$

¹I think a multiset (a.k.a. a bag) is like a normal set, but you can have multiple copies of the same thing in it.

Very rough sketch of δ_i :



There are two kinds of event:

- *Computation events*, where a process makes a transition
- *Delivery events*, where a message is delivered to a process

If an event E is possible in a configuration C , it may occur and result in a configuration C' : $C \xrightarrow{e} C'$.

Example

1. Suppose $C = (S_0, \dots, S_{11}, U)$ and $\delta_i(S_i) = (S'_i, out)$. Then $C \xrightarrow{comp(i)} C' = (S_0, \dots, S'_i, \dots, S_{11}, U \cup out)$.
2. Suppose $C = (S_0, \dots, S_{11}, U \cup \{m\})$ where $m = (P_j, d, P_i)$. Then $C \xrightarrow{del(m)} C' = (S_0, \dots, S_i, \dots, S_{11}, U)$ where if $S_i = (q, n, in)$ then $S'_i = (q, n, in \cup \{m\})$.

\mathbf{Q} has a nonempty subset \mathbf{Q}_{init} of initial control states, and a subset \mathbf{Q}_{final} of final control states. A state is *initial* (*final*) if its control state is. A configuration is *initial* (*final*) if the state of each of its processes is.

Example: $\mathbf{Q}_{init} = \{init\}$ and $\mathbf{Q}_{final} = \{leader, follower\}$.
Also, $\mathbf{S}_{init} = \mathbf{Q}_{init} \times \mathbf{N} \times \{\emptyset\}$ and $\mathbf{S}_{final} = \mathbf{Q}_{final} \times \mathbf{N} \times \mathcal{F}(\mathbf{M})$.
Also, $\mathbf{C}_{init} = (\mathbf{S}_{init})^{12} \times \mathcal{F}(\mathbf{M})$ and $\mathbf{C}_{final} = (\mathbf{S}_{final})^{12} \times \mathcal{F}(\mathbf{M})$.

In general, if δ is a transition function, S is a final state, and $\delta(S) = (S', out)$, then S' is final. Hence, if C is final and $C \xrightarrow{e} C'$, then C' is final.

An *evolution* of a distributed system described by a configuration C_0 is of the form $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} C_2 \xrightarrow{e_2} \dots$. An evolution $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \dots$ is *terminating* if there exists j such that C_j is final.

If C_0 describes an *asynchronous* system then an evolution $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \dots$ is *admissible* if:

1. For each process P_i there are infinitely many j such that $e_j = comp(i)$.
2. If $e_j = comp(i)$ and m is created in e_j , then $e_k = del(m)$ for some k .

In an admissible evolution of an asynchronous system, each process performs infinitely many transitions, and every message that is created is delivered. There need be no finite upper bound on the number of events between two transitions of a process or between the creation and the delivery of a message.

If C_0 describes a *synchronous* system then an evolution $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \dots$ is *admissible* if it consists of a sequence of *rounds*. In a round,

1. all outstanding messages are delivered, and then
2. each process performs a computation step which results in at most one message being created, addressed to any given process.

There is a fixed bound on the number of events between the creation and delivery of a message.

2 Leader Election

The *leader election problem* is to design a process that stores just two data – its name $n : \mathbf{N}$ and its status $s : \{\text{null}, \text{leader}, \text{follower}\}$ – so that for any $k > 1$, if S is a system consisting of k copies of P – that may differ only in storing different names – connected in a ring, and in which there are initially no undelivered messages, then every admissible evolution of S contains a final configuration in which the status of one P is *leader*, and the status of every other P is *follower*.

A. The synchronous case

Fix $k > 1$. Choose distinct n_0, \dots, n_{k-1} . P has variables s , initially *null*, and n , initially n_i in P_i . P 's transitions are described as follows, where c refers to the clockwise neighbour:

$$\begin{aligned}
 \textit{init} &= c!n \rightarrow \textit{active} \\
 \textit{leader} &= ?in \rightarrow \textit{leader} \\
 \textit{follower} &= ?in \rightarrow \textit{follower} \\
 \textit{active} &= ?in \rightarrow \textit{IF}(\\
 &\quad \textit{done} \in \textit{data}(in) : s := \textit{follower} \rightarrow c?\textit{done} \rightarrow \textit{follower} \\
 &\quad n \in \textit{data}(in) : s := \textit{leader} \rightarrow c!\textit{done} \rightarrow \textit{leader} \\
 &\quad n < \max(in) : c!\max(in) \rightarrow \textit{active} \\
 &\quad n > \max(in) : \textit{active}
 \end{aligned}$$

[Notes: (1) The IF construct means “if...then...elsif...then...elsif...then...” and so on. (2) An empty set doesn't actually have a maximum, to the IF wouldn't appear to cover the case where $in = \emptyset$. It works (I think) if you pretend that $\max(\emptyset) = -1$, or that the last branch will always be executed if $in = \emptyset$.]

Let $n^* = \max n_0, \dots, n_{k-1}$, and suppose $n^* = n_m$.

Lemma 1

Suppose $0 \leq r < k$.

- (a) In round r ,
 - (i) P_{m+r} sends n^*
 - (ii) If $j \neq m + r$ then P_j does not send n^*
 - (iii) No P_j sends *done*
 - (iv) If $j \neq m$ then no P_i with $i \in [m \dots j - 1]$ sends j
- (b)
 - (i) P_j is in control state *active*.
 - (ii) $s_j = \text{null}$

Lemma 2

Suppose $0 \leq r < k$.

- (a) In round $k + r$, P_{m+r} sends *done*.
- (b) At the end of round $k + r$,
 - (i) P_m is in control state *leader* and $s_m = \text{leader}$
 - (ii) If $j \in (m, m + r]$ then P_j is in control state *follower* and $s_j = \text{follower}$
 - (iii) If $j \in (m + r, m)$ then P_j is in control state *active* and $s_j = \text{null}$

Corollary 3

At the end of round $2k - 1$,

- (a) $s_m = \text{leader}$
- (b) If $j \neq m$ then $s_j = \text{follower}$

Message complexity

The number of messages created in the first $2k$ rounds is $O(k^2)$.

B. The asynchronous case

Consider the system S , but now viewed as being asynchronous.

Lemma 4

Suppose C is a configuration in an admissible evolution of S .

- (a) P_j is *leader* iff $s_j = \text{leader}$.
- (b) P_j is *follower* iff $s_j = \text{follower}$.
- (c) P_j is *init* or *active* iff $s_j = \text{null}$.

- (d) If C contains (P_i, n^*, P_{i+1}) then P_m is active and $s_j = \text{null}$ for $j \neq m$.
- (e) C contains at most one message of the form (P_i, n^*, P_{i+1}) .
- (f) If $j \neq m$ and $i \in [m \dots j - 1]$, then C does not contain (P_i, n_j, P_{i+1}) .
- (g) If $j \neq m$ then $s_j \neq \text{leader}$.
- (h) If C contains $(P_{m+j}, \text{done}, P_{m+j+1})$, where $d \leq j < k$, then $s_m = \text{leader}$ and $s_i = \text{follower}$ for $i \in (m, m + j]$ and P_i is *active* for $i \in (m + j, m)$.
- (i) C contains at most one message of the form $(P_i, \text{done}, P_{i+1})$.

Lemma 5

Suppose $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \dots$ is an admissible evolution of S .

- (a) If $0 \leq j < k$ then some C_h contains an undelivered message $(P_{m+j}, n^*, P_{m+j+1})$.
- (b) If $0 \leq j < k$ then in some C_h , $s_m = \text{leader}$ and $s_i = \text{follower}$ for $i \in (m, m + j]$ and there is an undelivered message $(P_{m+j}, \text{done}, P_{m+j+1})$

Corollary 6

Every admissible evolution of S contains a configuration in which $s_m = \text{leader}$ and $s_j = \text{follower}$ for $j \neq m$.

Further reading on Leader Election: Lynch ch. 3 and 15, Attiya & Welch ch. 3

3 Logical clocks and vector clocks

Suppose $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \dots$ is an evolution of an asynchronous system S consisting of n processes P_1, \dots, P_n . Assume that $e_i \neq e_j$ if $i \neq j$ and that no two messages created in the evolution are identical. Let e_0^i, e_1^i, \dots be the events performed by P_i (in order of occurrence in the evolution).

Definition

The *happens before* relation is the smallest transitive relation \rightarrow such that:

1. $e_j^i \rightarrow e_{j+1}^i$ for all i, j , and
2. if m is created in e and consumed in e' then $e \rightarrow e'$.

References Lynch 18, A&W 6, Coulouris 10, Mullender 4

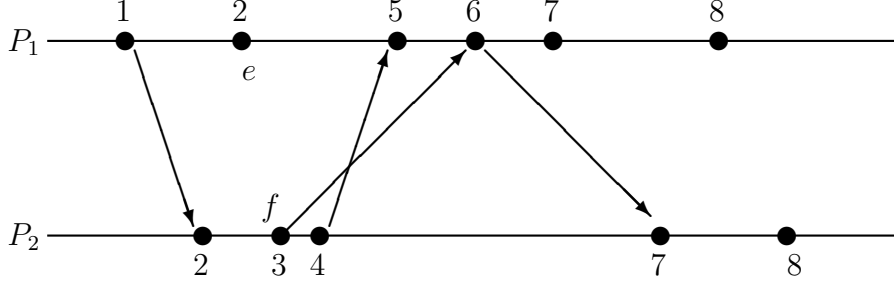
We derive an evolution of a system S' . Each P_i' has a logical clock, and integer variable c_i , initially 0. If P_i performs e_j then P_i' performs e_j' by carrying out the non-sending actions of E_j , then executing: $c_i := 1 + \max\{c_i, \max\{t | (P_j', (d, t), P_i') \text{ is consumed in } e_j'\}\}$, and then carrying out the sending actions of e_j , but sending (d, c_i) instead of d .

$\lambda(e_j)$ is defined to be the value of c_i when execution of e_j' finishes.

Lemma 7

If $e \rightarrow f$ is in the evolution of S then $\lambda(e) < \lambda(f)$. The converse does not hold, however.

Example (Send-Consume Diagram)



We derive an evolution of a system S'' . Each P_i'' has a vector clock, an array variable $v_i[1..n]$, initially $v_i[j] = 0$ for $1 \leq j \leq n$. If P performs e_h then P_i'' performs e_h'' by carrying out the non-sending actions of e_h , then executing

$$\begin{aligned} v_i[i] &:= v_i[i] + 1 \quad \text{and for } j \neq i, \\ v_i[j] &:= \max\{v_i[j], \max\{t_j | (P_k'', (d, \langle t_1..t_n \rangle), P_i'') \text{ is consumed in } e_h''\}\} \end{aligned}$$

and then carrying out the sending actions of e_h , but sending (d, v_i) , instead of d . $v(e_h)$ is defined to be the value of v_i when execution of e_h'' finishes.

Note: In any configuration of the evolution of S'' , $v_j[i] \leq v_i[i]$.

Define $v \leq w$ if $v[i] \leq w[i] \forall i$, and $v < w$ if $v \leq w$ and $v \neq w$. Also, v and w are *incomparable* if $v \not\leq w$ and $w \not\leq v$.

Lemma 8

$e \rightarrow f$ is the evolution of S iff $v(e) < v(f)$.

Proof (sketch)

\Rightarrow From the definitions.

\Leftarrow Suppose $e \not\rightarrow f$ where P_i performs e and P_j performs f . Then $v(f)[i] < v(e)[i]$ so $v(e) \not\leq v(f)$.

Corollary 9

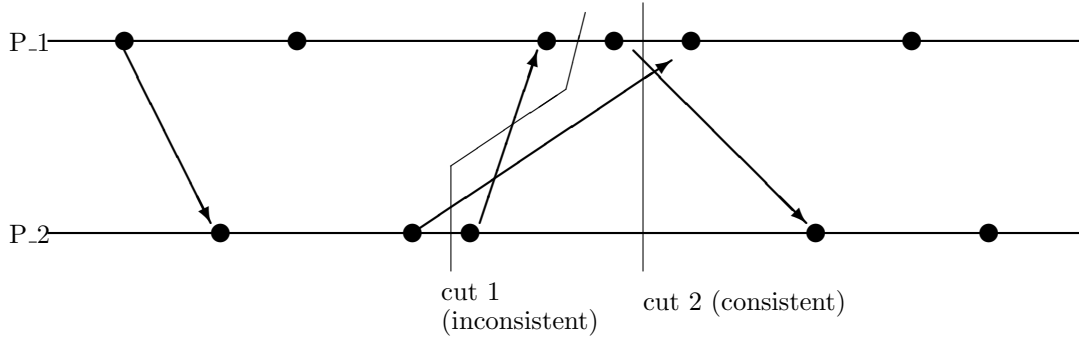
e and f are concurrent, i.e. $e \not\rightarrow f$ and $f \not\rightarrow e$, iff $v(e)$ and $v(f)$ are incomparable.

Suppose the links between the processes of S are FIFO queues. For convenience, assume that when an event involves the creation of a message m from P_i to P_j , it also involves the delivery of m to the FIFO queue q_{ij} , from which P_j can consume messages.

A *cut* is a (finite) subset of $\{e_0, e_1, \dots\}$ that is the union of the events in finite initial segments $e_0^i, \dots, e_{s(i)}^i$ for $1 \leq i \leq n$. A cut determines a configuration: each P_i is in the state after $e_{s(i)}^i$ has occurred. A cut K is consistent if $e \in K$ and $e' \rightarrow e$ implies $e' \in K$.

A configuration is consistent if it is the configuration determined by a consistent cut.

Example



The *Chandy, Lamport distributed snapshot algorithm* can be used to construct a consistent configuration in evolution of an asynchronous system.²

For concreteness, suppose P_1, \dots, P_n are bank processes that send one another amounts of money from accounts they maintain. Assume that if P_i and P_j are adjacent, then P_i sends infinitely many messages to P_j , and assume that each P_i consumes from each q_{ki} infinitely often.

Introduce a *monitor* process P_0 , which is directly connected to each P_i . P_0 acts as follows (repeatedly):

- send *take snapshot* to each P_i
- receive (*take snapshot* and) state from each P_i , and construct configuration.

The behaviour of $P_i (1 \leq i \leq n)$ is augmented as follows:

- $s := state$
- send *take snapshot* to each adjacent P_k
- $Q_{hi} := \langle \rangle$ (all $h \neq 0$ adjacent)
- $status_j := finished$
- $status_k := unfinished$ ($k \neq j$ adjacent)
- (without performing any money-transfer events)
- Resume executing money-transfer events.
- On consuming v from q_{ki} with $status_k = unfinished$, record v in Q_{ki} .
- On consuming *take snapshot* from q_{ki} , $status_k := finished$.
- If every $status_k = finished$, send s together with all Q_{ki} to P_0 , and cease recording.

Global snapshots can be used to detect if a *stable* property holds, e.g. if there is a deadlock in the system. (Suggested reading: Mullender ch. 4, Lynch ch. 19).

²Hooray.

4 Commit Protocols

Consider a synchronous distributed database, whose component processes can fail by stopping. To maintain consistency, no transaction can be *committed* by one process but *aborted* by another. More precisely, we consider first the *two-phase commit protocol*, which ensures that if no process fails then a configuration is reached in which every process fails then a configuration is reached in which every process has committed or every process has aborted. One process is deemed the *coordinator*.

In phase 1, the coordinator sets its status to the appropriate one of *can commit* and *must abort*, and each non-coordinator does likewise and sends its *status* to the coordinator. If the coordinator's status is *can commit* and it receives notification from each non-coordinator that *its status* is *can commit*, then the decision is *commit*; otherwise the decision is *abort*. In phase 2, the coordinator sends the decision to each non-coordinator, which acts on it.

Lemma 10

If in an admissible evolution of the two-phase commit protocol no process fails, then a configuration is reached in which every process has committed or every process has aborted.

The *three-phase commit protocol* modifies and extends the two-phase commit protocol to take account of the possibility that the coordinator may fail. It achieves the stronger property that a configuration is reached in which every process *that has not failed* has committed or aborted, and in which it is not the case that some process has committed and some [other] process has aborted.

Suggested reading: Lynch, section 7.