#40

Specifying Distributed Systems

Butler W. Lampson
Cambridge Research Laboratory
Digital Equipment Corporation
One Kendall Square
Cambridge, MA 02139

October 1988
In Constructive Methods in Computer Science, ed. M. Broy, NATO ASI Series F: Computer and Systems Sciences 55, Springer, 1989, pp 367-396

These notes describe a method for specifying concurrent and distributed systems, and illustrate it with a number of examples, mostly of storage systems. The specification method is due to Lamport (1983, 1988), and the notation is an extension due to Nelson (1987) of Dijkstra's (1976) guarded commands.

We begin by defining states and actions. Then we present the guarded command notation for composing actions, give an example, and define its semantics in several ways. Next we explain what we mean by a specification, and what it means for an implementation to satisfy a specification. A simple example illustrates these ideas.

The rest of the notes apply these ideas to specifications and implementations for a number of interesting concurrent systems:

Ordinary memory, with two implementations using caches;

Write buffered memory, which has a considerably weaker specification chosen to facilitate concurrent implementations;

Transactional memory, which has a weaker specification of a different kind chosen to facilitate fault-tolerant implementations;

Distributed memory, which has a yet weaker specification than buffered memory chosen to facilitate highly available implementations. We give a brief account of how to use this memory with a tree-structured address space in a highly available naming service.

Thread synchronization primitives.

States and actions

We describe a system as a state space, an initial point in the space, and a set of atomic actions which take a state into an *outcome*, either another state or the looping outcome, which we denote $\perp$. The state space is the cartesian product of subspaces called the *variables* or *state functions*,

NATO ASI Series, Vol. F 55
Constructive Methods in Computing Science
Edited by M. Broy

depending on whether we are thinking about a program or a specification. Some of the variables and actions are part of the system's *interface*.

Each action may be more or less arbitrarily classified as part of a *process*. The behavior of the system is determined by the rule that from state s the next state can be s' if there is any action that takes s to s'. Thus the computation is an arbitrary interleaving of actions from different processes.

Sometimes it is convenient to recognize the *program counter* of a process as part of the state. We will use the state functions:

at(α) true when the PC is at the start of operation α

in(α) true when the PC is at the start of any action in the operation α

after(α) true when the PC is immediately after some action in the operation α, but not in(α).

When the actions correspond to the statements of a program, these state components are essential, since the ways in which they can change reflect the flow of control between statements. The soda machine example below may help to clarify this point.

An atomic action can be viewed in several equivalent ways.

- A *transition* of the system from one state to another; any execution sequence can be described by an interleaving of these transitions.
- A *relation between states and outcomes*, i.e., a set of pairs: state, outcome. We usually define the relation by
  A so = P(s, o)
  If A contains (s, o) and (s, o'), $o \neq o'$, A is *non-deterministic*. If there is an s for which A contains no (s, o), A is *partial*.
- A *relation on predicates*, written {P} A {Q}
  If A s s' then P(s) ⇒ Q(s')
- A pair of *predicate transformers*: wp and wlp, such that
  wp(A, R) = wp(A, true) ∧ wlp(A, R)
  wlp(A, ?) distributes over any conjunction
  wp (A, ?) distributes over any non-empty conjunction

The connection between A as a relation and A as a predicate transformer is
  wp(A, R) s = every outcome of A from s satisfies R
  wlp(A, R) s = every proper outcome of A from s satisfies R
We abbreviate this with the single line
  w(l)p(A, R) s = every (proper) outcome of A from s satisfies R
Of course, the looping outcome doesn't satisfy any predicate.

Define the *guard* of A by
  $\mathcal{G}$(A) = ¬ wp(A, false) or $\mathcal{G}$(A) s = (∃ o: A so)
$\mathcal{G}$(A) is true in a state if A relates it to some outcome (which might be ⊥). If A is total, $\mathcal{G}$(A) = true.

We build up actions out of a few primitives, as well as an arbitrarily rich set of operators and datatypes which we won't describe in detail. The primitives are

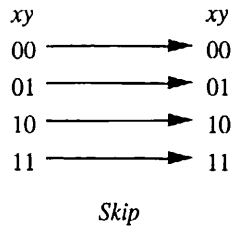
;	sequential composition
→	guard
□	or
⊠	else
|	variable introduction
if ... fi	
do ... od	

These are defined below in several equivalent ways: operationally, as relations between states, and as predicate transformers. We omit the relational and predicate-transformer definitions of **do**. For details see Dijkstra (1976) or Nelson (1987); the latter shows how to define **do** in terms of the other operators and recursion.

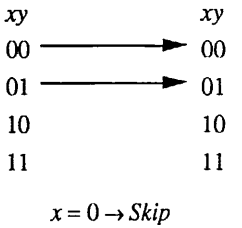
The precedence of operators is the same as their order in the list above; i.e., ";" binds most tightly and "|" least tightly.

Actions: operational definition (what the machine does)

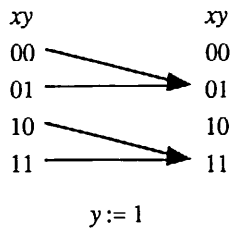
skip	do nothing
loop	loop indefinitely
fail	don't get here
(P→ A)	activate A from a state where P is true
(A □ B)	activate A or B
(A ⊠ B)	activate A, else B if A has no outcome
(A ; B)	activate A, then B
(**if** A **fi**)	activate A until it succeeds
(**do** A **od**)	activate A until it fails



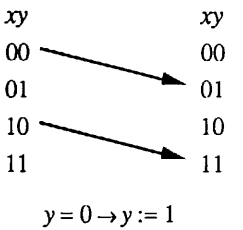
Skip



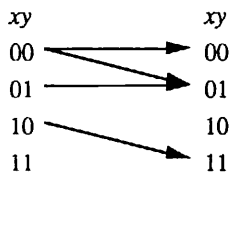
$x = 0 \rightarrow Skip$
(partial)



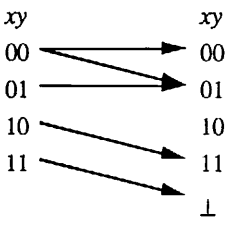
$y := 1$



$y = 0 \rightarrow y := 1$
(partial)



$x = 0 \rightarrow Skip$
$\square\ y = 0 \rightarrow y := 1$
(partial, non-deterministic)



if $x = 0 \rightarrow Skip$
$\square y = 0 \rightarrow y := 1$
fi
(non-deterministic)

Figure 1. The anatomy of a guarded command. The command in the lower right is composed of the subcommands shown in the rest of the figure.

Actions: relational definition

skip so $\equiv s = o$

loop so $\equiv o = \perp$

fail so $\equiv$ false

(P$\rightarrow$ A) so $\equiv$ P $s \wedge$ A so

(A $\square$ B) so $\equiv$ A $so \vee$ B so

(A $\boxtimes$ B) so $\equiv$ A $so \vee$ (B $so \wedge \neg \mathcal{G}$(A) s)

(A ; B) so $\equiv (\exists\, s'$: A $ss' \wedge$ B $s'o) \vee$ (A $so \wedge o = \perp$)

(if A **fi)** so $\equiv$ A $so \vee (\neg \mathcal{G}$(A) $s \wedge$ o $= \perp$)

(x := y) so $\equiv$ o is the same state as s, except that the x component equals y.

(x | A) so $\equiv (\forall\, s', o'$: proj$_x(s')$=$s \wedge$ proj$_x(o')$=$o \Rightarrow$ A $s'o'$), where proj$_x$ is the projection that drops the x component of the state, and takes $\perp$ to itself. Thus | is the operator for variable introduction.

See figure 1 for an example which shows the relations for various actions. Note that **if** A **fi** makes the relation defined by A total by relating states such that $\mathcal{G}$(A)=false to $\perp$.

The idiom x | P(x) $\rightarrow$ A can be read "With a new x such that P(x) do A".

Actions: predicate transformer definition

		$\mathcal{G}$(...) =
w(l)p(skip, R)	$\equiv$ R	true
w(l)p(loop, R)	$\equiv$ false(true)	true
w(l)p(fail, R)	$\equiv$ true	false
w(l)p(P$\rightarrow$ A, R)	$\equiv \neg$ P $\vee$ w(l)p(A, R)	P $\wedge\ \mathcal{G}$(A)
w(l)p(A $\square$ B, R)	$\equiv$ w(l)p(A, R) $\wedge$ w(l)p(B, R)	$\mathcal{G}$(A) $\vee\ \mathcal{G}$(B)
w(l)p(A $\boxtimes$ B, R)	$\equiv$ w(l)p(A, R) $\wedge$ ($\mathcal{G}$(A) $\vee$ w(l)p(B, R))	$\mathcal{G}$(A) $\vee\ \mathcal{G}$(B)
w(l)p(A ; B, R)	$\equiv$ w(l)p(A, w(l)p(B, R))	$\neg$wp(A, $\neg\mathcal{G}$(B))
w(l)p(x := y, R)	$\equiv$ R(x: y)	true
w(l)p(x | A, R)	$\equiv \forall\, x$: w(l)p(A, R)	$\exists\, x$: $\mathcal{G}$(A)
wp(**if** A **fi**, R)	$\equiv$ wp(A, R) $\wedge\ \mathcal{G}$(A)	true
wlp(**if** A **fi**, R)	$\equiv$ wlp(A, R)	true

Programs as specifications

Following Lamport (1988) we say that a specification consists of

A state space, the cartesian product of a set of variables or *state functions*, divided into *interface* and *internal* variables.

An initial value for the state.

A set of atomic actions, divided into *interface* and *internal* actions, with the possible state transitions for each action (the *transition axioms*).

A set of *liveness axioms*, written in some form of temporal logic. A treatment of liveness is beyond the scope of these notes.

An implementation I satisfies the specification S if:

The interface variables of S and I are the same, and have the same initial values.

There is a function F from the state of I to the internal state of S (the *abstraction function*) such that:

F takes the initial state of I to the initial internal state of S.

Every allowed transition of I when mapped by F is an allowed transition of S, or is the identity on S.

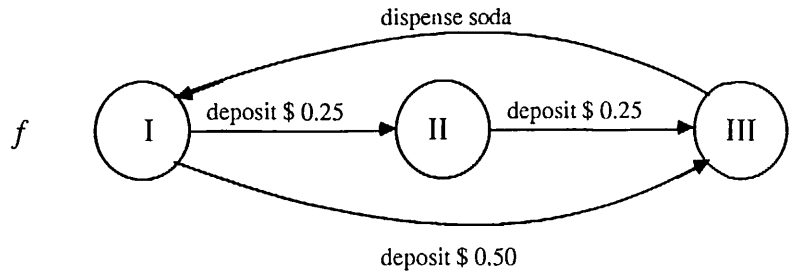
The transition and liveness axioms of I mapped by F imply the liveness axioms of S.

Soda machine

We give a simple example (due to Lamport) of a soda machine with two specifications: a transition diagram of the kind familiar from textbooks on finite state automata, and a program. It is not hard to show that the second one is an implementation of the first; the second is annotated on the right with the function F. The reverse is also true, but for reasons which are beyond the scope of these notes.

We indicate the interface variables and actions by underlining them.

Transition diagram specification



Program specification

interface depositCoin ...;
dispenseSoda ...;

var x : {0, 25, 50};
y : {25, 50};

			Abstraction function F
			if at(α)→ I
α:	**do**	⟨ $x := 0$ ⟩	
β:	;	**do** ⟨ $x < 50$ ⟩ →	□ at(β)→ **if** x=0 →
			□ x=25→
			□ x=50→
γ:		⟨ y := depositCoin	□ at(γ)→ **if** x=0 →
		; $x+y \leq 50$ → skip ⟩	□ x=25→
			□ x=50→
			fi
δ:		; ⟨ $x := x+y$ ⟩	□ at(δ)→ **if** $x+y$=25→
			□ $x+y$=50→
			□ $x+y$=75→
		od	**fi**
ε:	;	⟨ dispenseSoda ⟩	□ at(ε)→
	od		**fi**

Notation

In writing the specifications and implementations, we use a few fairly standard notations.

If T is a type, we write t, t', t_1 etc. for variables of type T.

If c_1, ..., c_n are constants, $\{c_1, ..., c_n\}$ is the enumeration type whose values are the c_i.

If T and U are types, $T \oplus U$ is the disjoint union of T and U. If c is a constant, we write $T \oplus c$ for $T \oplus \{c\}$.

If T and U are types, $T \to U$ is the type of functions from T to U; the overloading with the $\to$ of guarded commands is hard to avoid. If f is a function, we write $f(x)$ or $f[x]$ for an application of f, and $f[x] := y$ for the operation that changes $f[x]$ to y and leaves f the same elsewhere. If f is undefined at x, we write $f[x] = \bot$.

If T is a type, **sequence of** T is the type of all sequences of elements of T, including the empty sequence. T is a subtype of **sequence of** T. We write $s \parallel s'$ for the concatenation of two sequences, and Λ for the empty sequence.

$\langle$ A $\rangle$ is an atomic action with the same semantics as A in isolation.

Memory

Simple memory

Here is a specification for simple addressable memory.

type	A;	*address*
	D;	*data*
var	m : A $\to$ D;	*memory*
Read(a, **var** d)	= $\langle\, d := m[a] \,\rangle$	
Write(a, d)	= $\langle\, m[a] := d \,\rangle$	
Swap(a, d, **var** d')	= $\langle\, d' := m[a];\ m[a] := d \,\rangle$	

Cache memory

Now we look at an implementation that uses a write-back cache. The abstraction function is given after the variable declarations. This implementation maintains the invariant that the number of addresses at which c is defined is constant; for a hardware cache the motivation should be obvious. A real cache would maintain a more complicated invariant, perhaps bounding the number of addresses at which c is defined.

type	A;	*address*
	D;	*data*
var	m : A $\to$ D;	*main memory*
	c : A $\to$ D $\oplus \bot$;	*cache (partial)*

abstraction function

$m_{\text{simple}}[a] : d$	=	**if** $c[a] \neq \bot \to d := c[a]$ ▯ $d := m[a]$ **fi**
dirty(a): BOOL	=	$c[a] \neq \bot \wedge c[a] \neq m[a]$
FlushOne	=	$a \mid c[a] \neq \bot \to$ **do** *dirty*$[a] \to m[a] := c[a]$ **od**; $c[a] := \bot$
Load(a)	=	$\langle$ **do** $c[a] = \bot \to$ FlushOne; $c[a] := m[a]$ **od** $\rangle$
Read(a, **var** d)	=	Load(a); $\langle\, d := c[a] \,\rangle$
Write(a, d)	=	$\langle$ **if** $c[a] = \bot \to$ FlushOne ▯ skip **fi** $\rangle$; $\langle\, c[a] := d \,\rangle$
Swap (a, d, **var** d')	=	Load(a); $\langle\, d' := c[a];\ c[a] := d \,\rangle$

Coherent cache memory

Here is a more complex implementation, suitable for a multiprocessor in which each processor has its own write-back cache. We still want the system to behave like a single shared memory. Again, the abstraction function follows the variables. Correctness depends on the invariant at the end. This implementation is some distance from being practical; in particular, a practical one would have *shared* and *dirty* as variables, with invariants relating their values to the definitions given here.

We write c_p instead of $c[p]$ for readability.

type A; *address*
D; *data*
P; *processor*

var $m : A \to D$; *main memory*
$c : P \to A \to D \oplus \bot$; *caches (partial)*

abstraction function

$m_{simple}[a] : d$ = **if** $\exists p: c_p[a] \neq \bot \to d := c_p[a]$
[] $d := m[a]$
fi

shared(*a*): BOOL = $\exists p, q: c_p[a] \neq \bot \wedge c_q[a] \neq \bot \wedge p \neq q$

dirty (*a*): BOOL = $\exists p: \; c_p[a] \neq \bot \wedge c_p[a] \neq m[a]$

Load(*p*, *a*) = **do** $c_p[a] = \bot \to$
FlushOne(*p*)
; **if** $\langle q \mid c_q[a] \neq \bot \to \; c_p[a] := c_q[a] \rangle$
[] $\langle c_p[a] := m[a] \rangle$
fi
od

FlushOne(*p*) = $a \mid c_p[a] \neq \bot \to$
$\langle$ **do** $\neg$*shared*[*a*] $\wedge$ *dirty*[*a*] $\to m[a] := c_p[a]$ **od** $\rangle$
; $c_p[a] := \bot$

Read(*p*, *a*, **var** *d*) = Load(*p*, *a*); $\langle d := c_p[a] \rangle$

Write(*p*, *a*, *d*) = **if** $c_p[a] = \bot \to$ FlushOne(*p*) [] skip **fi**
; $\langle \; c_p[a] := d$
; **do** $q \mid c_q[a] \neq \bot \wedge c_q[a] \neq c_p[a] \to c_q[a] := c_p[a]$ **od**
$\rangle$

Swap (*p*, *a*, *d*, **var** *d* ') = Load(*p*, *a*); $\langle d' := c_p[a]$; Write(*p*, *a*, *d*) $\rangle$

Invariant

$c_p[a] \neq \bot \wedge c_q[a] \neq \bot \Rightarrow c_p[a] = c_q[a]$

Write-buffered memory

We now turn to a memory with a different specification. This is *not* another implementation of the simple memory specification. In this memory, each processor sees its own writes as it would in a simple memory, but it sees only a non-deterministic sampling of the writes done by other processors. A *FlushAll* operation is added to permit synchronization.

The motivation for using the weaker specification is the possibility of building a faster processor if writes don't have to be synchronized as closely as is required by the simple memory specification. After giving the specification, we show how to implement a critical section within which variables shared with other processor can be read and written with the same results that the simple memory would give.

type A; *address*
D; *data*
P; *processor*

var $m : A \to D$; *main memory*
$b : P \to A \to D \oplus \bot$; *buffers (partial)*

Flush(*p*, *a*) = $b_p[a] \neq \bot \to \langle m[a] := b_p[a] \rangle ; \langle b_p[a] := \bot \rangle$

FlushSome = *p*, *a* | Flush(*p*, *a*); FlushSome
[] skip

Read(*p*, *a*, **var** *d*) = **if** $\langle \; b_p[a] = \bot \to \; d := m[a] \rangle$
[] $\langle q \mid b_q[a] \neq \bot \to \; d := b_q[a] \rangle$
fi

Write(*p*, *a*, *d*) = $\langle b_p[a] := d \rangle$

Swap (*p*, *a*, *d*, **var** *d'*) = FlushSome; $\langle d' := m[a]; m[a] := d \rangle$

FlushAll(*p*) = **do** *a* | Flush(*p*, *a*) **od**

Critical section

We want to get the effect of an ordinary critical section using simple memory, so we write that as a specification (the right-hand column below). The implementation (the left-hand column below) uses buffered memory to achieve the same effect. Provided the non-critical section doesn't reference the memory and the critical section doesn't reference the lock, a program using buffered memory with the left-hand implementation of mutual exclusion has the same semantics as (as a relation, is a subset of) the same program using simple memory with the standard right-hand implementation of mutual exclusion. To nest critical sections we use the usual device: partition A into disjoint subsets, each protected by a different lock.

var m : $A \to D$;
b : $P \to A \to D \oplus \bot$;

const l := *the address of a location to be used as a lock*

abstraction function

$m_{simple}[a] : d$ = **if** $p \mid b_p[a] \neq \bot \to$ $d := p[a]$
▣ $d := m[a]$
fi

for $p \in P$

Implementation (using buffered memory)		*Specification (using simple memory)*
do d_p |		**do** d_p |
α_p:	$\langle\, d_p := 1 \,\rangle$	$\langle\, d_p := 1 \,\rangle$
β_p:	; **do** $\langle\, d_p \neq 0 \,\rangle \to$	; **do** $\langle\, d_p \neq 0 \,\rangle \to$
γ_p:	Swap(p, l, 1, d_p)	Swap(l, 1, d_p)
	od	**od**
δ_p:	; *critical section*	; *critical section*
ε_p:	; FlushAll(p)	
κ_p:	; Write(p, l, 0)	; Write(l, 0)
λ_p:	; *non-critical section*	; *non-critical section*
od		**od**

initially $\forall\, p, a : b_p[a] = \bot\,,\ m[l] = 0$

assume

$A \in |\lambda_p| \Rightarrow$ A independent of m : no Read, Write or Swap in A

$A \in |\delta_p| \Rightarrow$ A independent of $m[l]$: no Read, Write or Swap(p, l, ...) in A

The proof depends on the following invariants for the implementation.

Invariants

(1) $CS_p \Rightarrow \quad (\neg\, CS_q \vee p = q\,)$
$\wedge\ m\,[l] \neq 0$
$\wedge\ b_q[l] \neq 0$
where $CS_p = \text{in}(\delta\varepsilon\kappa_p) \vee (\ \text{at}(\beta_p) \wedge d_p = 0\ )$

(2) $\neg\, \text{in}(\delta\varepsilon_p) \wedge\ a \neq l \Rightarrow b_p[a] = \bot$

Multiple write-buffered memory

This version is still weaker, since each processor keeps a sequence of all its writes to each location rather than just the last one. Again, the motivation is to allow a higher-performance implementation, by increasing the amount of buffering at the expense of more non-determinism. The same critical section works.

type A; *address*
D; *data*
P; *processor*
E = **sequence of** D;

var $m : A \to D$; *main memory*
b : $P \to A \to E$; *buffers*

Flush(p, a) = $d, e \mid b_p[a] = d \parallel e \to \langle\, m[a] := d \,\rangle\,;\, \langle\, b_p[a] := e \,\rangle$

FlushSome = $p, a \mid$ Flush(p, a); FlushSome
□ skip

Read(p, a): d = **if** $\langle\ b_p[a] = \Lambda \to d := m[a] \,\rangle$
□ $\langle\, q, e_1, d', e_2 \mid\ b_q[a] = e_1 \parallel d' \parallel e_2$
$\wedge\ (q \neq p \vee e_2 = \Lambda) \to d := d' \,\rangle$
fi

Write(p, a, d) = $\langle\, b_p[a] := b_p[a] \parallel d \,\rangle$

Swap (p, a, d): d' = FlushSome; $\langle\, d' := m[a];\ m[a] := d \,\rangle$

FlushAll(p) = **do** a \| Flush(p, a) **od**

Transactions

This example describes the characteristics of a memory that provides *transactions* so that several writes can be done atomically with respect to failure and restart of the memory. The idea is that the memory is not obliged to remember the writes of a transaction until it has accepted the transaction's *commit*; until then it may discard the writes and indicate that the transaction has aborted.

A real transaction system also provides atomicity with respect to concurrent accesses by other transactions, but this elaboration is beyond the scope of these notes.

We write $\text{Proc}_t(...)$ for $\text{Proc}(t, ...)$ and l_t for $l[t]$.

type A; *address*
D; *data*
T; *transaction*
X = {ok, abort};

var $m : A \to D$; *memory*
$b : T \to A \to D$; *backup*

Abort = $\langle\, m := b_t \,\rangle$; $x :=$ abort

$\underline{\text{Begin}}_t()$ = $\langle\, b_t := m \,\rangle$

$\underline{\text{Read}}_t(a,$ **var** $d,$ **var** $x)$ = $\langle\, d := m[a] \,\rangle$; $x :=$ ok
[] Abort

$\underline{\text{Write}}_t(a, d,$ **var** $x)$ = $\langle\, m[a]:=d \,\rangle$; $x :=$ ok
[] Abort

$\underline{\text{Commit}}_t($**var** $x)$ = $x :=$ ok
[] Abort

Undo implementation

This is one of the standard implementations of the specification above: the old memory values are remembered, and restored in case of an abort.

var $m : A \to D$; *memory*
$l : T \to A \to D \oplus \perp$; *log*

abstraction function

$b_t[a]: d$ = **if** $l_t[a] \neq \perp \to d := l_t[a]$
[] $d := m[a]$
fi

m_{simple} = m

$\text{Abort}_t()$ = $\langle$ **do** $a \mid l_t[a] \neq \perp \to m[a] := l_t[a];\ l_t[a] := \perp$ **od** $\rangle$; $x :=$ abort

$\underline{\text{Begin}}_t()$ = **do** $a \mid l_t[a] \neq \perp \to \langle\, l_t[a] := \perp \,\rangle$ **od**

$\underline{\text{Read}}_t(a,$ **var** $d,$ **var** $x)$ = $\langle\, d := m[a] \,\rangle$; $x :=$ ok
[] Abort

$\underline{\text{Write}}_t(a, d,$ **var** $x)$ = **do** $l_t[a] = \perp \to \langle\, l_t[a] := m[a] \,\rangle$ **od**; $\langle\, m[a]:=d \,\rangle$; $x :=$ ok
[] Abort

$\underline{\text{Commit}}_t($**var** $x)$ = $x :=$ ok
[] Abort

Compare this abstraction function with the one for the cache memory.

Redo implementation

This is the other standard implementation: the writes are remembered and done in the memory only at commit time. Essentially the same work is done as in the undo version, but in different places; notice how similar the code sequences are.

var $m : A \to D$; *memory*
$l : T \to A \to D \oplus \perp$; *log*

abstraction function

b_t = m

$m_{\text{simple}}[a]: d$ = **if** $t \mid l_t[a] \neq \perp \to$ $d := l_t[a]$
[] $d := m[a]$
fi

Abort = $x :=$ abort

$\underline{\text{Begin}}_t()$ = **do** $a \mid l_t[a] \neq \perp \to \langle\, l_t[a] := \perp \,\rangle$ **od**

$\underline{\text{Read}}_t(a,$ **var** d , **var** $x)$ = **if** $l_t[a] \neq \perp \to d := l_t[a]$ [] $\langle\, d := m[a] \,\rangle$ **fi**; $x :=$ ok
[] Abort

$\underline{\text{Write}}_t(a, d,$ **var** $x)$ = $l_t[a] := d$; $x :=$ ok
[] Abort

$\underline{\text{Commit}}_t($**var** $x)$: = $\langle$ **do** $a \mid l_t[a] \neq \perp \to m[a] := l_t[a];\ l_t[a] := \perp$ **od** $\rangle$; $x :=$ ok
[] Abort

Undo version with non-atomic abort

Note the atomicity of commit in the redo version and abort in the undo version; a real implementation gets this with a commit record, instead of using a large atomic action. Here is how it goes for the undo version.

var m : $A \to D$; *memory*
l : $T \to A \to D \oplus \bot$; *log*
ab : $T \to BOOL$; *aborted*

abstraction function

$b_t[a]: d$ = **if** $l_t[a] \neq \bot \to d := l_t[a]$
☒ $d := m[a]$
fi

$m_{simple}[a]: d$ = **if** $t \mid ab_t \wedge l_t[a] \neq \bot \to d := l_t[a]$
☒ $d := m[a]$
fi

$Abort_t()$ = $\langle\, ab_t := true \,\rangle$
; **do** $a \mid \langle\, l_t[a] \neq \bot \,\rangle \to \langle\, m[a] := l_t[a] \,\rangle ; \langle\, l_t[a] := \bot \,\rangle$ **od**
; $x :=$ abort

$\underline{Begin_t}()$ = $ab_t :=$ false; **do** $a \mid l_t[a] \neq \bot \to \langle\, l_t[a] := \bot \,\rangle$ **od**

$\underline{Read_t}(a,$ **var** $d,$ **var** $x)$ = $\neg ab_t \to \langle\, d := m[a] \,\rangle; x :=$ ok
☐ Abort

$\underline{Write_t}(a, d,$ **var** $x)$ = $\neg ab_t \to$ **do** $l_t[a] = \bot \to \langle\, l_t[a] := m[a] \,\rangle$ **od**; $\langle\, m[a]:=d \,\rangle; x :=$ ok
☐ Abort

$\underline{Commit_t}($**var** $x)$ = $\neg ab_t \to x :=$ ok
☐ Abort

Name service

This section describes a tree-structured storage system which was designed as the basis of a large-scale, highly-available distributed name service. After explaining the properties of the service informally, we give specifications of the essential abstractions that underlie it.

A name service maps a name for an entity (an individual, organization or service) into a set of labeled properties, each of which is a string. Typical properties are

password=XQE$#

mailboxes={Cabernet, Zinfandel}

network address=173#4456#1655476653

distribution list={Birrell, Needham, Schroeder}

A name service is not a general database: the set of names changes slowly, and the properties given name also change slowly. Furthermore, the integrity constraints of a useful name servic are much weaker those of a database. Nor is it like a file directory system, which must create look up names much faster than a name service, but need not be as large or as available. Eithe database or a file system root can be named by the name service, however.

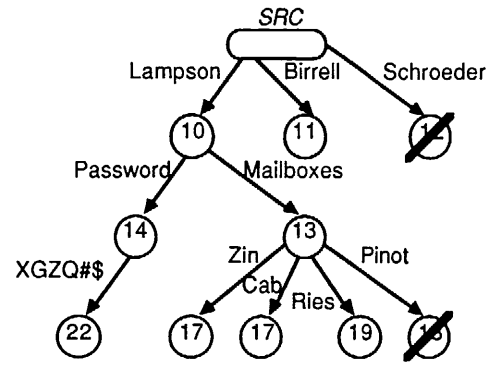


Figure 2: The tree of directory values

A directory is not simply a mapping from simple names to values. Instead, it contains a *tree* o values (see Figure 2). An arc of the tree carries a *name* (N), which is just a string, written nex the arc in the figure. A node carries a *timestamp* (S), represented by a number in the figure, a *mark* which is either *present* or *absent*. Absent nodes are struck through in the figure. A path through the tree is defined by a sequence of names (A); we write this sequence in the Unix sty e.g., Lampson/Password. For the value of the path there are three interesting cases:

- If the path a/n ends in a leaf that is an only child, we say that n is the value of a. This rule applies to the path Lampson/Password/XGZQ#$3, and hence we say that XGZQ#$3 is value of Lampson/Password.
- If the path a/n_i ends in a leaf that is not an only child, and its siblings are labeled $n_1 ... n_k$, we say that the set $\{n_1 ... n_k\}$ is the value of a. For example, {Zin, Cab, Ries, Pinot} is the val of Lampson/Mailboxes.
- If the path a does not end in a leaf, we say that the subtree rooted in the node where it ends i the value of a. For example, the value of Lampson is the subtree rooted in the node with timestamp 10.

An update to a directory makes the node at the end of a given path present or absent. The upda is timestamped, and a later timestamp takes precedence over an earlier one with the same path.

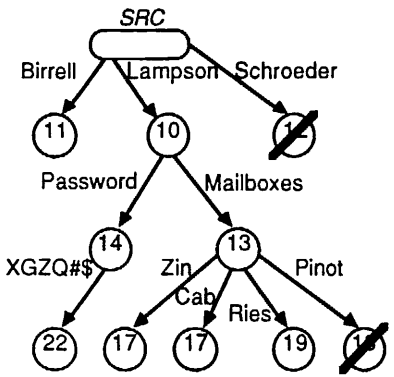
The subtleties of this scheme are discussed later; its purpose is to allow the tree to be updated concurrently from a number of places without any prior synchronization.

A value is determined by the sequence of update operations which have been applied to an initial empty value. An update can be thought of as a function that takes one value into another. Suppose the update functions have the following properties:

- Total: it always makes sense to apply an update function.
- Commutative: the order in which two updates are applied does not affect the result.
- Idempotent: applying the same update twice has the same effect as applying it once.

Then it follows that the *set* of updates that have been applied uniquely defines the state of the value.

It can be shown that the updates on values defined earlier are total, commutative and idempotent. Hence a set of updates uniquely defines a value. This observation is the basis of the concurrency control scheme for the name service. The right side of Figure 3 gives one sequence of updates which will produce the value on the left.




```
P Lampson:4/Password:11/UIO&6Z:12
P Lampson:10
P Birrell:11
A Schroeder:12
P Lampson:10/Mailboxes:13
P Lampson:10/Password:14
P Lampson:10/Mailboxes:13/Zin:17
P Lampson:10/Mailboxes:13/Cab:17
A Lampson:10/Mailboxes:13/Pinot:18
P Lampson:10/Mailboxes:13/Ries:19
P Lampson:10/Password:14/XGZQ#$:22
```

Figure 3: A possible sequence of updates

The presence of the timestamps at each name in the path ensures that the update is modifying the value that the client intended. This is significant when two clients concurrently try to create the same name. The two updates will have different timestamps, and the earlier one will lose. The fact that later modifications, e.g. to set the password, include the creation timestamp ensures that those made by the earlier client will also lose. Without the timestamps there would be no way to tell them apart, and the final value might be a mixture of the two sets of updates.

The client sees a single name service, and is not concerned with the actual machines on which it is implemented or the replication of the database which makes it reliable. The administrator allocates resources to the implementation of the service and reconfigures it to deal with long-term failures. Instead of a single directory, he sees a set of *directory copies* (DC) stored in different servers. Figure 4 shows this situation for the *DEC*/SRC directory, which is stored on four servers named alpha, beta, gamma, and delta. A directory reference now includes a list of the servers that store its DCs. A lookup can try one or more of the servers to find a copy from which to read.

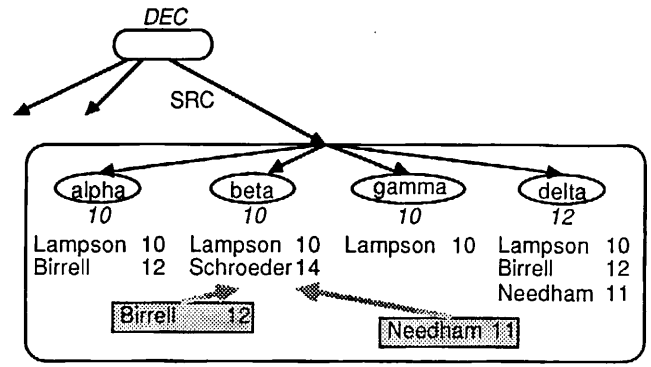


Figure 4: Directory copies

The copies are kept approximately, but not exactly the same. The figure shows four updates to *SRC*, with timestamps 10, 11, 12 and 14. The copy on delta is current to time 12, as indicate by the italic *12* under it, called its *lastSweep* field. The others have different sets of updates, bu are current only to time 10. Each copy also has a *nextS* value which is the next timestamp it wil assign to an update originating there; this value can only increase.

An update originates at one DC, and is initially recorded there. The basic method for spreading updates to all the copies is a *sweep* operation, which visits every DC, collects a complete set of updates, and then writes this set to every DC. The sweep has a timestamp *sweepS*, and before reads from a DC it increases that DC's *nextS* to *sweepS*; this ensures that the sweep collects all updates earlier than *sweepS*. After writing to a DC, the sweep sets that DC's *lastSweep* to *sweepS*. Figure 5 shows the state of *SRC* after a sweep at time 14.

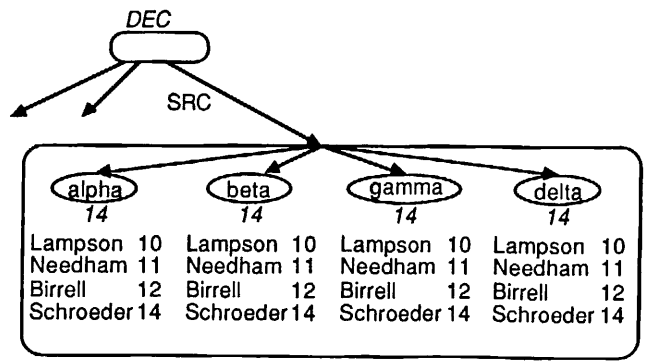


Figure 5: The directory after a Sweep

In order to speed up the spreading of updates, any DC may send some updates to any other DC in a message. Figure 4 shows the updates for Birrell and Needham being sent to server beta. Most updates should be distributed in messages, but it is extremely difficult to make this method fully reliable. The sweep, on the other hand, is quite easy to implement reliably.

A sweep's major problem is to obtain the set of DCs reliably. The set of servers stored in the parent is not suitable, because it is too difficult to ensure that the sweep gets a complete set if the directory's parent or the set of DCs is changing during the sweep. Instead, all the DCs are linked into a *ring*, shown by the fat arrows in figure 6. Each arrow represents the name of the server to which it points. The sweep starts at any DC and follows the arrows; if it eventually reaches the starting point, then it has found a complete set of DCs. Of course, this operation need not be done sequentially; given a hint about the contents of the set, say from the parent, the sweep can visit all the DCs and read out the ring pointers concurrently.

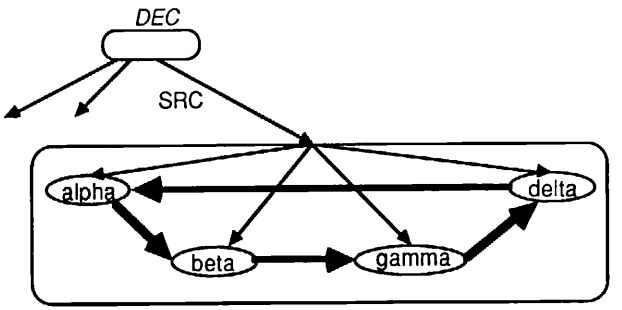


Figure 6: The ring of directory copies

DCs can be added or removed by straightforward splicing of the ring. If a server fails permanently, however (say it gets blown up), or if the set of servers is partitioned by a network failure that lasts for a long time, the ring must be reformed. In the process, an update will be lost if it originated in a server that is not in the new ring and has not been distributed. The ring is reformed by starting a new *epoch* for the directory and building a new ring from scratch, using the DR or information provided by the administrator about which servers should be included. An epoch is identified by a timestamp, and the most recent epoch that has ever had a complete ring is the one that defines the contents of the directory. Once the new epoch's ring has been successfully completed, the ring pointers for older epochs can be removed. Since starting a new epoch may change the database, it is never done automatically, but must be controlled by an administrator.

Distributed writes

Here is the abstraction for the name service's update semantics. The details of the tree of values are deferred until later; this specification depends only on the fact that updates are total, commutative and idempotent. We begin with a specification that says nothing about multiple copies; this is the client's view of the name service. Compare this with the write-buffered memory.

type	V;	*value*
	U = V→ V;	*update, assumed total, commutative, and idempotent*
	W = **set of** U;	*updates "in progress"*
var	m : V;	*memory*
	b : W;	*buffer*
AddSome(**var** v)	= $u \mid u \in b \wedge u(v) \neq v \rightarrow v := u(v)$; AddSome($v$)	
	□ skip	
Read(**var** v)	= $\langle v := m$; AddSome(v) $\rangle$	
Update(u)	= $\langle b := b \cup \{u\} \rangle$	
Sweep()	= $\langle$ **do** $u \mid u \in b \rightarrow m := u(m)$; $b := b - \{u\}$ **od** $\rangle$	

Update and Sweep were called Write and Flush in the specification for buffered writes. This differs in that there is no ordering on b, there are no updates in b that a Read is guaranteed to see, and there is no Swap operation.

You might think that Sweep is too atomic, and that it should be written to move one u from b to m in each atomic action. However, if two systems have the same $b \cup m$, the one with the smaller b is an implementation of the one with the larger b, so a system with non-atomic Sweep implements a specification with atomic Sweep.

We can substitute distinguishable for idempotent and ordered for commutative as properties of updates. AddSome and Sweep must be changed to apply the updates in order. If the updates are ordered, and we require that Update's argument follows any update already in m, then the boundary between m and b can be defined by the last update in m. This is a conveneint way to summarize the information in b about how much of the state can be read deterministically. In the name server application the updates are ordered by their timestamps, and the boundary is called *last Sweep*.

N-copy version

Now for an implementation that makes the copies visible. It would be neater to give each copy its own version of m and its own set b of recent updates. However, this makes it quite difficult to define the abstraction function. Instead, we simply give each copy its version of b, and define m to be the result of starting with an initial value v_0 and applying all the updates known to every copy. To this end the auxiliary function apply turns a set of updates w into a value v by applying all of them to v_0.

type	V;	*value*
	U = V→ V;	*update, assumed total, commutative, and idempotent*
	W = **set of** U;	*updates "in progress"*
	P;	*processor*
var	b : P→ W;	*buffers*

Abstraction function

m_{simple} = $apply(\bigcap_{p \in P} b[p])$

b_{simple} = $\bigcup_{p \in P} b[p] - \bigcap_{p \in P} b[p]$

In other words, the abstract m is all the updates that every processor has, and the abstract b is all the updates known to some processor but not to all of them.

apply(w): v = $v := v_0$; **do** $u \mid u \in w \rightarrow v' := u(v); w := w - \{u\}$ **od**

Read(p, **var** v) = $v := apply(b[p])$

Update(p, u) = $\langle\, b[p] := b[p] \cup \{u\} \,\rangle$

Sweep() = $w \mid \langle\, w := \bigcup_{p \in P} b[p] \,\rangle$

; **do** $p,u \mid u \in w \wedge u \notin b[p] \rightarrow \langle\, b[p] := b[p] \cup \{u\} \,\rangle$ **od**

Since this meant to be viewed as an implementation, we have given the least atomic Sweep, rather than the most atomic one. Abstractly an update moves from b to m when it is added to the last processor that didn't have it already.

Tree memory

Finally, we show the abstraction for the tree-structured memory that the name service needs. To be used with the distributed writes specification, the updates must be timestamped so that they can be ordered. This detail is omitted here in order to focus attention on the basic idea.

We use the notation $x \leftarrow y$ for $x \neq y \rightarrow x := y$. This allows us to copy a tree from v' to v with the idiom

do $a \mid v[a] \leftarrow v'[a]$ **od**

which changes the function v to agree with v' at every point. Recall also that || stands for concatenation of sequences; we use sequences of names as addresses here, and often need to concatenate such path names.

type	N;	*name*
	D;	*data*
	A = **sequence of** N;	*address*
	V = A → D ⊕ ⊥;	*tree value*
var	m : V;	*memory*

Read(a, **var** v) = $\langle$ **do** $a' \mid v[a'] \leftarrow m[a \parallel a']$ **od** $\rangle$

Write(a, v) = $\langle$ **do** $a' \mid m[a \parallel a'] \leftarrow v[a']$ **od** $\rangle$

Write(a, d) = $v \mid \forall a: v[a] = \perp \rightarrow v[\Lambda] = d$; Write(a, v)

Read copies the subtree of m rooted at a to v. Write(a, v) makes the subtree of m rooted at a equal to v. Write(a, d) sets $m[a]$ to d and makes undefined the rest of the subtree rooted at m.

Timestamped tree memory

We now introduce timestamps on the writes, in fact more of them that are needed to provide write ordering. The name service uses timestamps at each node in the tree to provide a poor man's transactions: each point in the memory is identified not only by the a that leads to it, but also by the timestamps of the writes that created the path to a. Thus conflicting use of the same names can be detected; the use with later timestamps will win. Figure 3 above shows an example.

We show only the write of a single value at a node identified by a given timestamped address b. The write fails (returning false in x) unless the timestamps of all the nodes on the path to node b match the ones in b. We write $m[a].d$ and $m[a].s$ for the d and s components of $m[a]$.

type	N;	*name*
	D;	*data*
	S;	*timestamp*
	A = **sequence of** N;	*address*
	V = A → (D × S) ⊕ ⊥;	*tree value*
	B = **sequence of** (N × S);	*address with timestamps*
var	m : V;	*memory*
Read(a, **var** v)	= ⟨ **do** a' | $v[a'] \leftarrow m[a \parallel a']$ **od** ⟩	
Write(b, d, **var** x)	= ⟨ a | for all $i \le$ length(b): $a[i] = b[i].n \rightarrow$	
	if for all $0<i<$length(b), $m[a[1..i]].s = b[i].s \rightarrow$	
	do a' | $m[a \parallel a'] \leftarrow \perp$ **od**	
	; $m[a] := (d, b[\text{length}(b)].s)$	
	; $x :=$ true	
	☒ $x :=$ false	
	fi	
	⟩	

The ordering relation on writes needed by the distributed writes specification is determined by the timestamped address:

$b_1 < b_2 = \exists\, i < \text{length}(b_1)$: $j<i \Rightarrow b_1[j]=b_2[j] \wedge b_1[i].n=b_2[i].n \wedge b_1[i].s<b_2[i].s$

In other words, $b_1<b_2$ if they match exactly up to some point, they have the same name at that point, and b_1 has the smaller timestamp at that point. This rule ensures that a write to a node near the root takes precedence over later writes into the subtree rooted at that node with an earlier timestamp. For example, Lampson:10 takes precedence over Lampson:4/Password:11.

Threads

The specification below for thread (or process) synchronization primitives is transcribed from (Birrell 1987), where it was expressed in the Larch specification language. Except for alerts, tl constructs should be familiar, although in some cases the meaning varies slightly from the liter ture. A condition variable is a substitute for busy waiting: a process waits there until a *Broadca* is done to the condition, or enough *Signals*. An *alert* is an indication to a thread that it should look around; it is delivered only after an *AlertWait*. Thus a thread which computes indefinitely without ever waiting on a condition or executing *TestAlert* will not notice the alert.

type	T;	*thread*
	M = T ⊕ nil;	*mutex*
	S = {busy, free};	*semaphore*
	C = **set of** T;	*condition*
var	a : **set of** T;	*alerted threads*
	self: T	*the thread doing the operation*
Acquire(**var** m)	= ⟨ m = nil → m := self ⟩	
Release(**var** m)	= ⟨ **if** $m \neq$ self → chaos	
	☒ m := nil	
	fi ⟩	
Wait(**var** m, **var** c)	= ⟨ **if** $m \neq$ self → chaos	
	☒ $c := c \cup$ {self}; m := nil	
	fi ⟩	
	; ⟨ m = nil ∧ ¬ self ∈ $c \rightarrow m$:= self ⟩	
Signal(**var** c)	= ⟨ **if** c = {} → skip	
	□ c' | $c \supset c' \rightarrow c := c'$	
	fi	
	⟩	
Broadcast(**var** c)	= ⟨ c := {} ⟩	
P(**var** s)	= ⟨ s=free → s := busy ⟩	
V(**var** s)	= ⟨ s := free ⟩	
Alert(t)	= ⟨ $a := a \cup \{t\}$ ⟩	
TestAlert(): b	= ⟨ b := (self ∈ a); $a := a -$ {self} ⟩	

```
AlertP(var s): b        = ⟨  s = free     → s := busy;      b := false
                          □  self ∈ a     → a := a – {self}; b := true
                          ⟩

AlertWait(var m, var c): b
                        = if m ≠ self → chaos
                          ⊠ ⟨  c := c ∪ {self}; m := nil ⟩
                          fi
                          ; ⟨  m = nil →
                                  m := self
                               ;     ¬ self ∈ c  → b := false
                                  □    self ∈ a  → b := true
                                                 ; c := c – {self}
                                                 ; a := a – {self}
                            ⟩
```

For comparison, we give the original Larch version of Wait:

```
type Condition = set of Thread initially {}

procedure Wait(var m: Mutex; var c: Condition)
      = composition of Enqueue, Resume end
   requires m = self
   modifies at most [ m, c ]

   atomic action Enqueue
   ensures (c_post=insert(c, self)) ∧ (m_post = nil)

   atomic action Resume
   when (m = nil) ∧ ¬(self ∈ c)
   ensures m_post = self & unchanged [ c ]
```

References

A. Birrell et. al. (1987). Synchronization primitives for a multiprocessor: A formal specification *ACM Operating Systems Review* **21**(5): 94-102.

E. Dijkstra (1976). *A Discipline of Programming*. Prentice-Hall.

L. Lamport (1988). A simple approach to specifying concurrent systems. Technical report 15 (revised), DEC Systems Research Center, Palo Alto. To appear in *Comm. ACM*, 1988.

L. Lamport (1983). Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, **5**(2): 190-222.

L. Lamport and F. Schneider (1984). The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, **6**(2): 281-296.

B. Lampson (1986). Designing a global name service. *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, pp 1-10.

G. Nelson (1987). A generalization of Dijkstra's calculus. Technical report 16, DEC Systems Research Center, Palo Alto.