# Principles for Computer System Design

Butler Lampson

We have learned depressingly little in the last ten years about how to build computer systems. But we have learned something about how to do the job more precisely, by writing more precise specifications, and by showing more precisely that an implementation meets its specification. Methods for doing this are of both intellectual and practical interest. I will explain the most useful such method and illustrate it with two examples:

> Connection establishment: Sending a reliable message over an unreliable network.
>
> Transactions: Making a large atomic action out of a sequence of small ones.

# Principles for Computer System Design

**10 years ago:** *Hints for Computer System Design*

**Not that much learned since then—disappointing**

> *Instead of standing on each other's shoulders, we stand on each other's toes.* (Hamming)

**One new thing: How to build systems more precisely**

> *If you think systems are expensive, try chaos.*

# Collaborators

| | |
|---|---|
| **Bob Taylor** | |
| **Chuck Thacker** | Workstations: Alto, Dorado, Firefly<br>Networks: AN1, AN2 |
| **Charles Simonyi** | Bravo WYSIWYG editor |
| **Nancy Lynch** | Reliable messages |
| **Howard Sturgis** | Transactions |
| **Martin Abadi**<br>**Mike Burrows**<br>**Morrie Gasser**<br>**Andy Goldstein**<br>**Charlie Kaufman**<br>**Ted Wobber** | Security |

# From Interfaces to Specifications

**Make modularity precise**

*Divide and conquer (Roman motto)*

*Design*

*Correctness*

*Documentation*

**Do it recursively**

*Any idea is better when made recursive (Randell)*

*Refinement*: One man’s implementation is another man’s spec. (*adapted from Perlis*)

*Composition*: Use actions from one spec in another.

# Specifying a System with State

**A *safety* property: nothing bad ever happens**
**Defined by a *state machine*:**

*state*: a set of values, usually divided into named *variables*

*actions*: named changes in the state

**A *liveness* property: something good eventually happens**

**These define *behavior*: all the possible sequence of actions**

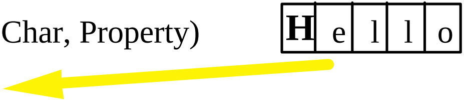
**Examples of systems with state:**

Data abstractions
Concurrent systems
Distributed systems

You can’t observe the actual state of the system from outside.
All you can see is the results of actions.

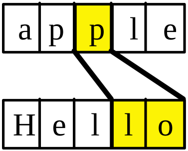
# Editable Formatted Text

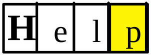
**State** *text*: sequence of (Char, Property)



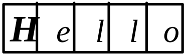
**Actions** *get(*2) returns (‘e’, (Times-Roman, ...))

*replace(*3, 5, 2, 3, a p p l e )





*look*(0, 5, *italic* := true)



This interface was used in the Bravo editor.
The implementation was about 20k lines of code.

# How to Write a Spec

**Figure out what the state is**

Choose it to make the spec clear, not to match the code.

**Describe the actions**

What they do to the state
What they return

**Helpful hints**

Notation is important; it helps you to think about what's going on.

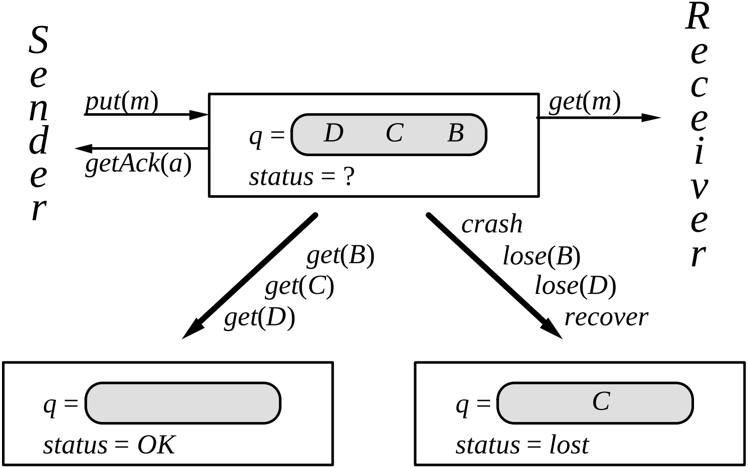
Invent a suitable vocabulary.

Fewer actions are better. *Less is more.*

More non-determinism is better; it allows more implementations.

> *I'm sorry I wrote you such a long letter; I didn't have time to write a short one.* *(Pascal)*

# Reliable Messages



*Sender*

*Receiver*

*put*(*m*)

*getAck*(*a*)

*get*(*m*)

*q* = *D* *C* *B*

*status* = ?

*get*(*B*)
*get*(*C*)
*get*(*D*)

*crash*
*lose*(*B*)
*lose*(*D*)
*recover*

*q* =
*status* = *OK*

*q* = *C*
*status* = *lost*

# Spec for Reliable Messages

$q$ : sequence$[M]$ := < >
$status$ : $\{OK, lost, ?\}$ := $lost$
$rec_{s/r}$ : Boolean := $false$ (short for ‘recovering’)

| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| $**put(m)$ | | append $m$ to $q$,<br>$status := ?$ | $*get(m)$ | $m$ first on $q$ | remove head of $q$,<br>if $q = \langle\rangle$, $status = ?$<br>then $status := OK$ |
| $*getAck(a)$ | $status = a$ | $status := lost$ | | | |
| $lose$ | $rec_s$ or $rec_r$ | delete some element from $q$;<br>if it’s the last then $status := lost$,<br>or $status := lost$ | | | |

# What “Implements” Means?

**Divide actions into** *external* **and** *internal***.**

**Y implements X if**

every external behavior of Y is an external behavior of X, and

Y’s liveness property implies X’s liveness property.

This expresses the idea that Y implements X if
you can’t tell Y apart from X by looking only at the external actions.

# Proving that Y implements X

Define an *abstraction function* *f* from the state of Y to the state of X.

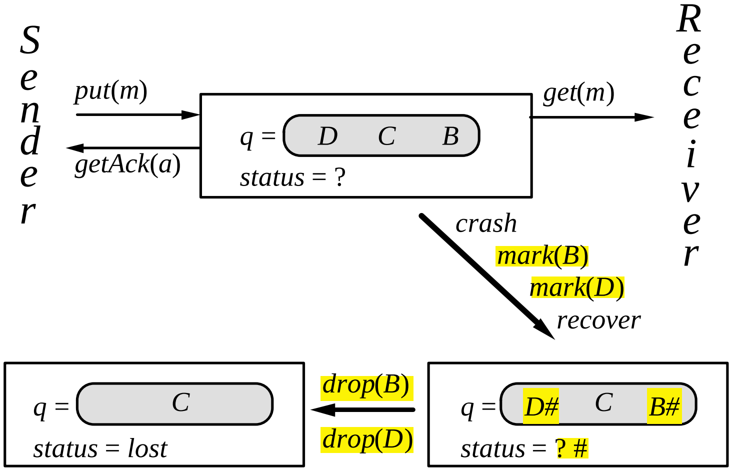
Show that Y *simulates* X:

1) *f* maps initial states of Y to initial states of X.
2) For each Y-action and each state *y*  
there is a sequence of X-actions that is the same externally,  
such that the diagram commutes.

$$\begin{tikzcd}
f(y) \arrow[r, "\textbf{X-actions}"] & f(y') \\
y \arrow[u, "f"] \arrow[r, "\textbf{Y-action}"] & y' \arrow[u, "f"]
\end{tikzcd}$$

This always works!

# Delayed-Decision Spec: Example



*Sender*

*put*(*m*)

*getAck*(*a*)

*q* = *D* *C* *B*

*status* = ?

*get*(*m*)

*Receiver*

*crash*

*mark*(*B*)

*mark*(*D*)

*recover*

*q* = *D#* *C* *B#*

*status* = ? #

*drop*(*B*)

*drop*(*D*)

*q* = *C*

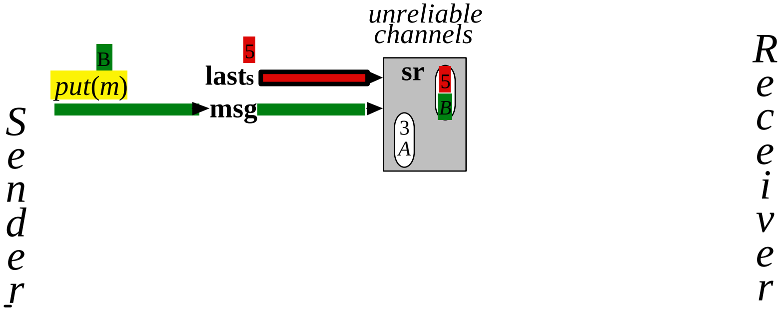
*status* = *lost*

The implementer wants the spec as non-deterministic as possible, to give him more freedom and make it easier to show correctness.

# A Generic Protocol G (1)

**Sender**
**actions state**

**Receiver**
**state actions**



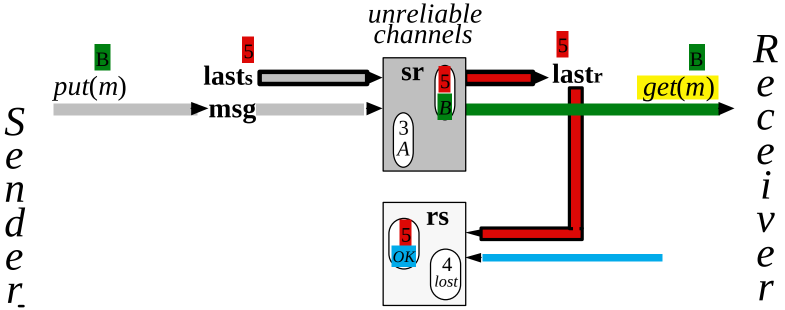
# A Generic Protocol G (2)

**Sender**

**actions** **state**

**Receiver**

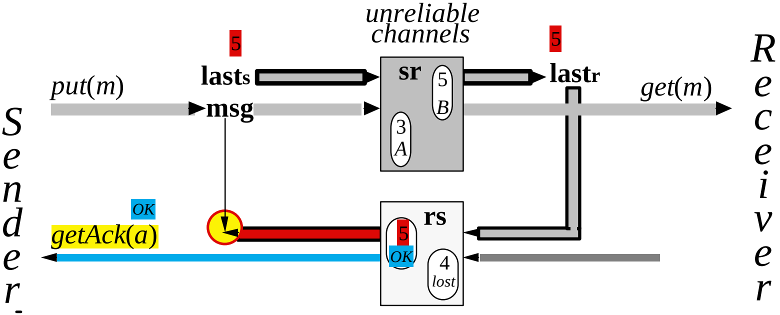
**state** **actions**



# A Generic Protocol G (3)

**Sender**
**actions** **state**

**Receiver**
**state** **actions**

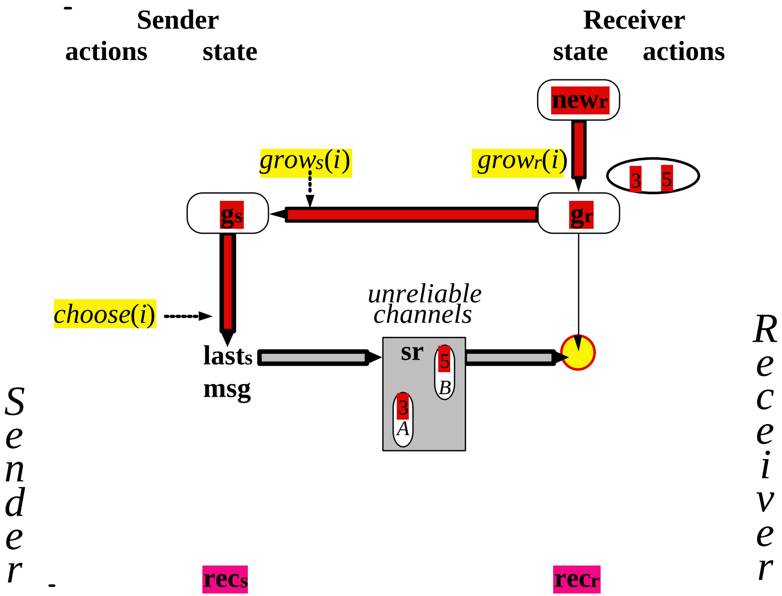


*unreliable channels*

**sr**

**rs**

# A Generic Protocol G (4)



**Sender**

**actions** **state**

**Receiver**

**state** **actions**

**new$_r$**

*grow$_s$(i)*

*grow$_r$(i)*

3 5

**g$_s$**

**g$_r$**

*choose(i)*

*unreliable channels*

**last$_s$**

**msg**

**sr**

5 *B*

3 *A*

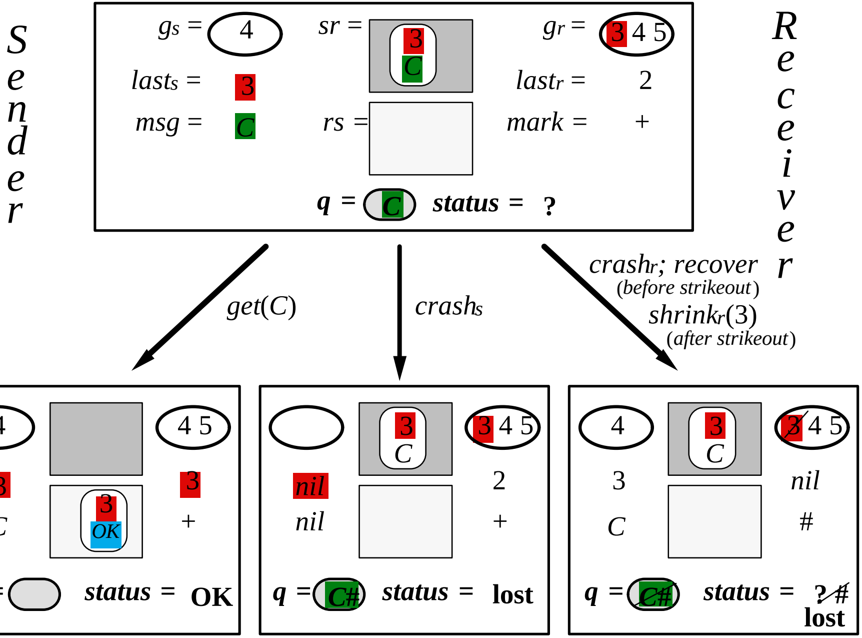
*Sender*

*Receiver*

**rec$_s$**

**rec$_r$**

# G at Work



Sender

$g_s$ = 4

$last_s$ = 3

msg = C

sr = 3 C

rs =

$g_r$ = 3 4 5

$last_r$ = 2

mark = +

**q** = C **status** = **?**

Receiver

get(C)

$crash_s$

$crash_r$; recover (before strikeout)

$shrink_r(3)$ (after strikeout)

4

3

C

3 OK

4 5

3

\+

**q** = **status** = **OK**

nil

nil

3 C

3 4 5

2

\+

**q** = C# **status** = **lost**

4

3

C

3 C

~~3~~ 4 5

nil

\#

**q** = ~~C#~~ **status** = **?** ~~#~~ **lost**

# Abstraction Function for G

*cur-q* = <*msg*> *if msg* *nil* and ($last_s = nil$ or $last_s \in g_r$)
< > otherwise

*old-q* = the messages in *sr* with *i*’s that are good and not = $last_s$

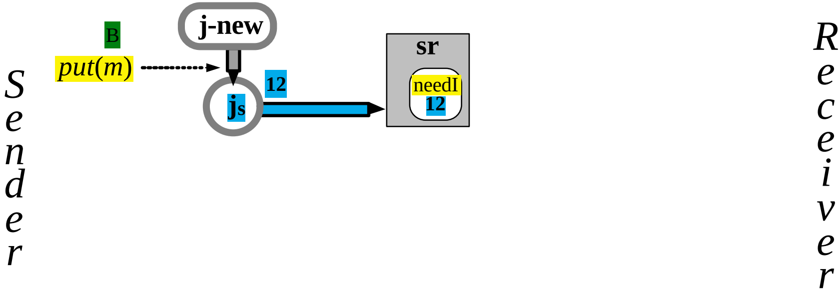
**q** *old-q* + *cur-q*

**status**□ ? if *cur-q* < >
*OK* if $last_s = last_r$ *nil*
*lost* if $last_s \notin (g_r \cup \{last_r\})$ or $last_s = nil$

$\boldsymbol{rec_{s/r}}$ $rec_{s/r}$

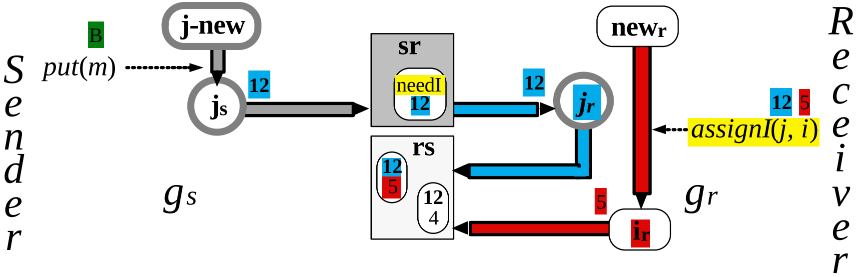
# The Handshake Protocol H (1)

-

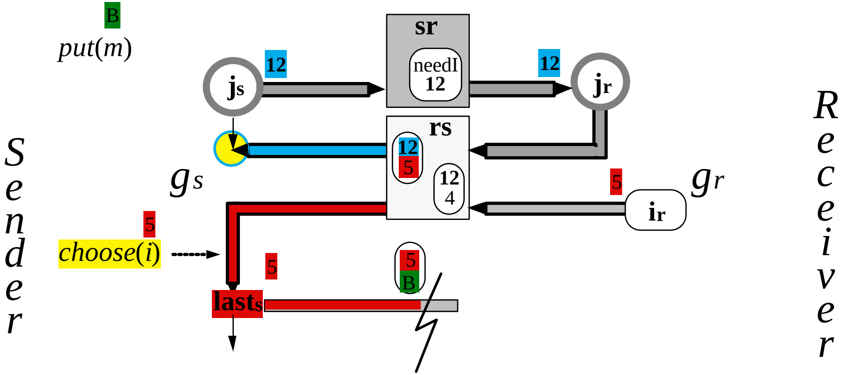


-

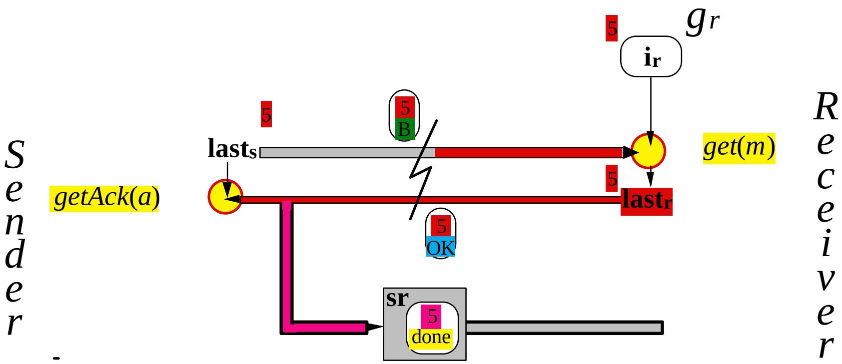
# The Handshake Protocol H (2)



# The Handshake Protocol H (3)

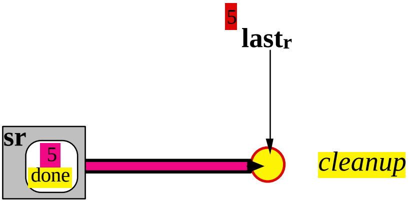


# The Handshake Protocol H (4)



# The Handshake Protocol H (5)

*Sender*



5

$\mathbf{last}_r$

**sr**

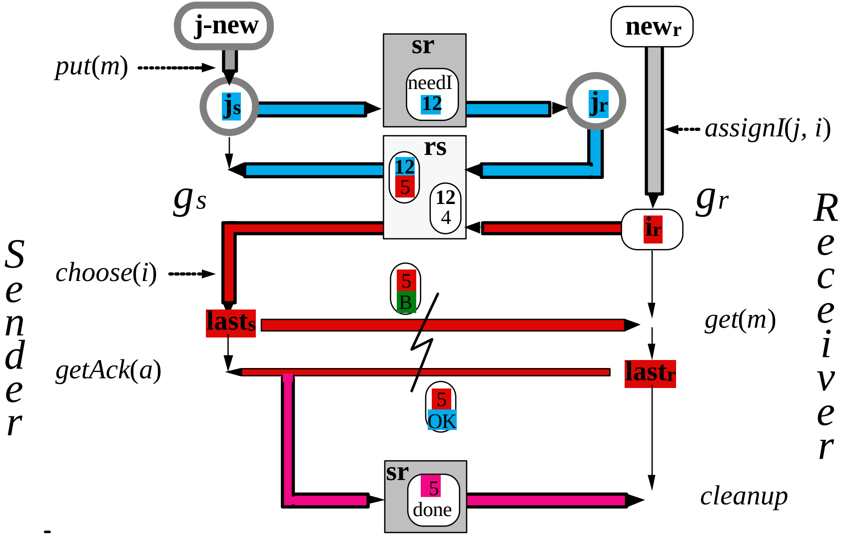
5

done

*cleanup*

*Receiver*

# The Handshake Protocol H (6)



# Abstraction Function for H

| G | H |
|---|---|
| $g_s$ | the $i$'s with $(j_s, i)$ in $rs$ |
| $g_r$ | $\{i_r\} - \{nil\}$ |
| $sr$ and $rs$ | the $(I, M)$ and $(I, A)$ messages in $sr$ and $rs$ |

$new_{s/r}$, $last_{s/r}$, and $msg$ are the same in G and H

| | |
|---|---|
| $grow_r(i)$ | receiver sets $i_r$ to an identifier from $new_r$ |
| $grow_s(i)$ | receiver sends $(j_s, i)$ |
| $shrink_s(i)$ | channel $rs$ loses the last copy of $(j_s, i)$ |
| $shrink_r(i)$ | receiver gets $(i_r$, done) |

> *An efficient program is an exercise in logical brinksmanship.*
> *(Dijkstra)*

# Reliable Messages: Summary

**Ideas**

Identifiers on messages

Sets of good identifiers, sender’s $\subseteq$ receiver’s

Cleanup

**The spec is simple.**

**Implementations are subtle because of crashes.**

The abstraction functions reveal their secrets.

The subtlety can be factored in a precise way.

# Atomic Actions

$S$ : *State*

| **Name** | **Guard** | **Effect** |
|---|---|---|
| $do(a):Val$ |  | $(S, val) := a(S)$ |

$X \quad Y$

5 5

$do(x := x - 1)$

4 5

$do(y := y + 1)$

4 6

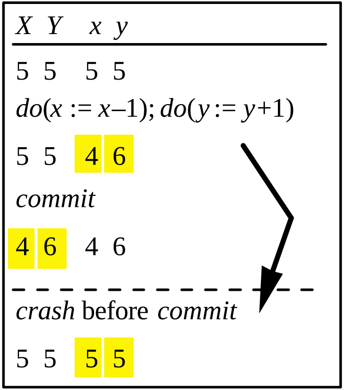
*A distributed system is a system in which I can’t get my work done because a computer has failed that I’ve never even heard of.*

*(Lamport)*

# Transactions: One Action at a Time

$S$ , $s$ : *State*

| **Name** | **Guard** | **Effect** |
|---|---|---|
| $do(a){:}Val$ | | $(s, val) := a(s)$ |
| *commit* | | $S := s$ |
| *crash* | | $s := S$ |



$X$ $Y$ $x$ $y$

5 5 5 5

$do(x := x-1); do(y := y+1)$

5 5 4 6

*commit*

4 6 4 6

*crash* before *commit*

5 5 5 5

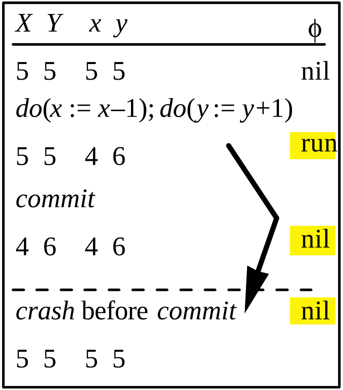
# Server Failures

$S$ , $s$ : $State$

$\phi$ : {nil, run} := nil

| Name | Guard | Effect |
|---|---|---|
| *begin* | $\phi$ = nil | $\phi$ := run |
| *do*($a$):*Val* | $\phi$ = run | ($s$, $val$) := $a$($s$) |
| *commit* | $\phi$ = run | $S$ := $s$, $\phi$ := nil |
| *crash* | | $s$ := $S$, $\phi$ := nil |

Note that we clean up the auxiliary state $\phi$.



$X$ $Y$ $x$ $y$ $\phi$

5 5 5 5 nil

*do*($x$ := $x$–1); *do*($y$ := $y$+1)

5 5 4 6 run

*commit*

4 6 4 6 nil

*crash* before *commit* nil

5 5 5 5

# Incremental State Changes: Logs (1)

$S$ , $s$ : *State*

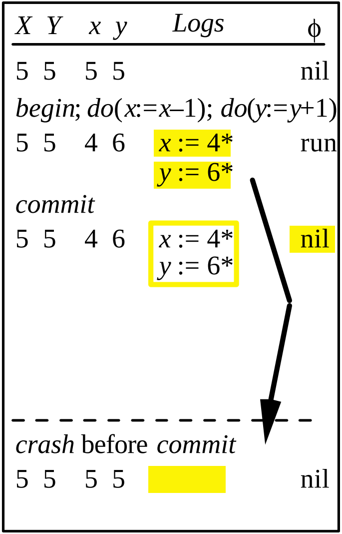
$L$ , $l$ : SEQ *Action* := < >

$\phi$ : {nil, run} := nil

$S = S + L$

$s, \phi = s, \phi$

| Name | Guard | Effect |
|---|---|---|
| *begin* | $\phi$ = nil | $\phi$ := run |
| *do*($a$):*Val* | $\phi$ = run | ($s$, *val*) := $a$($s$), $l$ +:= $a$ |
| *commit* | $\phi$ = run | $L$ := $l$, $\phi$ := nil |
| . . . | | |
| *crash* | | $l$ := $L$, $s$ := $S$+$L$, $\phi$:=nil |



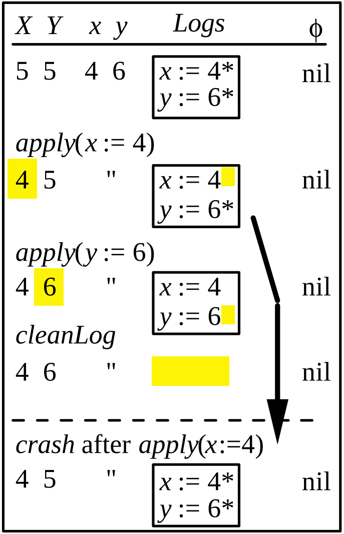
| $X$ | $Y$ | $x$ | $y$ | *Logs* | $\phi$ |
|---|---|---|---|---|---|
| 5 | 5 | 5 | 5 | | nil |
| *begin*; *do*($x$:=$x$–1); *do*($y$:=$y$+1) | | | | | |
| 5 | 5 | 4 | 6 | $x$ := 4*<br>$y$ := 6* | run |
| *commit* | | | | | |
| 5 | 5 | 4 | 6 | $x$ := 4*<br>$y$ := 6* | nil |
| *crash* before *commit* | | | | | |
| 5 | 5 | 5 | 5 | | nil |

# Incremental State Changes: Logs (2)

*S* , *s* : *State*
*L* , *l* : SEQ *Action*
ϕ : {nil, run}

*S* = *S* + *L*
s, ϕ = s, ϕ

| Name | Guard | Effect |
|---|---|---|
| *begin*, *do*, and *commit* as before | | |
| *apply*(*a*) | *a* = head(*l*) | *S* := *S* + *a*, *l* := tail(*l*) |
| *cleanLog* | *L* in *S* | *L* := < > |
| *crash* | | *l* := *L*, *s* := *S*+*L*, ϕ:=nil |



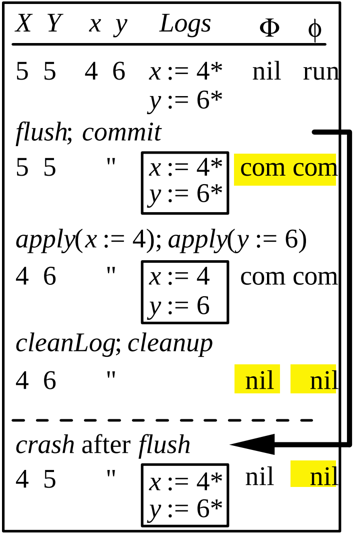
| *X* | *Y* | *x* | *y* | *Logs* | ϕ |
|---|---|---|---|---|---|
| 5 | 5 | 4 | 6 | *x* := 4*<br>*y* := 6* | nil |
| *apply*(*x* := 4) | | | | | |
| 4 | 5 | " | | *x* := 4<br>*y* := 6* | nil |
| *apply*(*y* := 6) | | | | | |
| 4 | 6 | " | | *x* := 4<br>*y* := 6 | nil |
| *cleanLog* | | | | | |
| 4 | 6 | " | | | nil |
| *crash* after *apply*(*x*:=4) | | | | | |
| 4 | 5 | " | | *x* := 4*<br>*y* := 6* | nil |

# Incremental Log Changes

$S$ , $s$ : *State*
$L$ , $l$ : SEQ *Action*
$\Phi$ , $\phi$ : {nil, run*, commit}

$L = L$ if $\phi$ = com else < >
$\phi = \phi$ if $\phi$   com else nil

| Name | Guard | Effect |
|---|---|---|
| *begin* and *do* as before | | |
| *flush* | $\phi$ = run | copy some of $l$ to $L$ |
| *commit* | $\phi$ = run, $L = l$ | $\Phi := \phi :=$ commit |
| *apply*($a$) | $\phi$ = commit, " | " |
| *cleanLog* | head($L$) in $S$<br>or $\phi$ = nil | $L$ := tail($L$) |
| *cleanup* | $L$ = < > | $\Phi := \phi :=$ nil |
| *crash* | | $l$ := < > if $\Phi$ = nil else $L$;<br>$s := S + l$, $\phi := \Phi$ |



| $X$ | $Y$ | $x$ | $y$ | *Logs* | $\Phi$ | $\phi$ |
|---|---|---|---|---|---|---|
| 5 | 5 | 4 | 6 | $x := 4*$<br>$y := 6*$ | nil | run |
| *flush*; *commit* | | | | | | |
| 5 | 5 | " | | $x := 4*$<br>$y := 6*$ | com | com |
| *apply*($x := 4$); *apply*($y := 6$) | | | | | | |
| 4 | 6 | " | | $x := 4$<br>$y := 6$ | com | com |
| *cleanLog*; *cleanup* | | | | | | |
| 4 | 6 | " | | | nil | nil |
| *crash* after *flush* | | | | | | |
| 4 | 5 | " | | $x := 4*$<br>$y := 6*$ | nil | nil |

# Distributed State and Log

$S_i$ , $s_i$ : *State*
$L_i$ , $l_i$ : SEQ *Action*
$\Phi_i$ , $\phi_i$ : {nil, run*, commit}
$S$, $L$, $\Phi$ are the products of the $S_i$, $L_i$, $\Phi_i$

$\phi$ = run if all $\phi_i$ = run
com if any $\phi_i$ = com
and any $L_i$ < >
nil otherwise

| Name | Guard | Effect |
|---|---|---|
| *begin* and *do* as before | | |
| *flush*$_i$ | $\phi_i$ = run | copy some of $l_i$ to $L_i$ |
| *prepare*$_i$ | $\phi_i$ = run, $L_i = l_i$ | $\Phi_i$ := run |
| *commit* | $\phi$ = run, $L = l$ | some $\Phi_i$ := $\phi_i$ := commit |
| *cleanLog* and *cleanup* as before | | |

*crash*$_i$ $l_i$ := < > if $\Phi_i$ = nil else $L_i$;
$s_i := S_i + l_i$, $\phi_i := \Phi_i$

# High Availability

**The Φ = commit is a possible single point of failure.**

With the usual two-phase commit (2PC) this is indeed a limitation on availability.

If data is replicated, an unreplicated commit is a weakness.

**Deal with this by using a highly available** *consensus* **algorithm for Φ.**

Lamport’s Paxos algorithm is the best currently known.

# Transactions: Summary

**Ideas**

- Logs
- Commit records
- Stable writes at critical points: prepare and commit
- Lazy cleanup

**The spec is simple.**

**Implementations are subtle because of crashes.**

- The abstraction functions reveal their secrets.
- The subtlety can be added one step at a time.

# How to Write a Spec

**Figure out what the state is**

Choose it to make the spec clear, not to match the code.

**Describe the actions**

What they do to the state
What they return

**Helpful hints**

Notation is important; it helps you to think about what's going on.

Invent a suitable vocabulary.

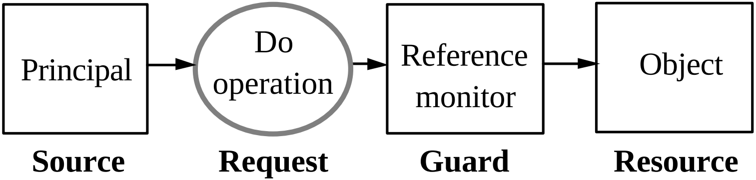
Fewer actions are better. *Less is more.*

More non-determinism is better; it allows more implementations.

*I'm sorry I wrote you such a long letter; I didn't have time to write a short one.* *(Pascal)*

# Security: The Access Control Model

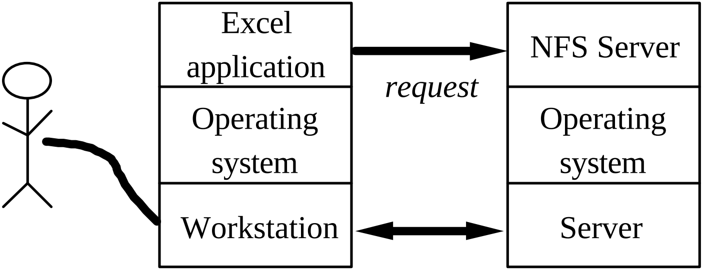
**Guards control access to valued resources.**



**Rules control the operations allowed**
for each principal and object.

| *Principal* may do | *Operation* on | *Object* |
|---|---|---|
| Taylor | Read | File “Raises” |
| Jones | Pay invoice 4325 | Account Q34 |
| Schwarzkopf | Fire three rounds | Bow gun |

# A Distributed System



# Principals

**Authentication:** **Who sent a message?**

**Authorization:** **Who is trusted?**

**Principal — abstraction of "who":**

| | |
|---|---|
| People | Lampson, Taylor |
| Machines | VaxSN12648, Jumbo |
| Services | SRC-NFS, X-server |
| Groups | SRC, DEC-Employees |
| Channels | Key #7438 |

# Theory of Principals

**Principal says statement** $\boxed{P\ \textbf{says}\ s}$

Lampson **says** “read /SRC/Lampson/foo”

SRC-CA **says** “Lampson’s key is #7438”

**Principal** *A* **speaks for** *B* $\boxed{A \Rightarrow B}$

If *A* says something, *B* says it too. So *A* is stronger than *B*.

**A secure channel:**

says things directly $\boxed{C\ \textbf{says}\ s}$

If *P* is the only sender on C $\boxed{C \Rightarrow P}$

**Examples**

Lampson => SRC

Key #7438 => Lampson

# Handing Off Authority

**Handoff rule:** If *A* **says** *B* => *A* then *B* => *A*

Reasonable if *A* is competent and accessible.

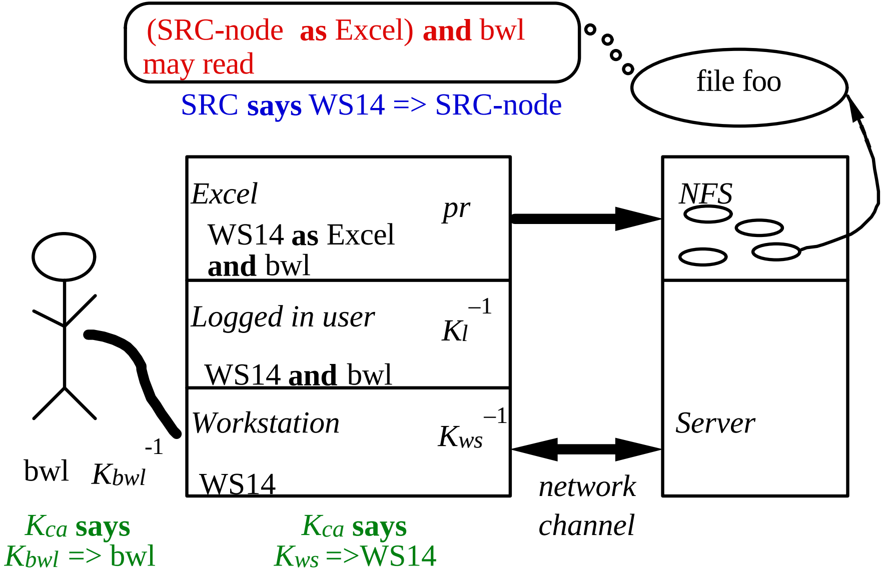
**Examples:**

SRC **says** Lampson **=>** SRC

Node key **says** Channel key **=>** Node key

*Any problem in computer science can be solved*
*with another level of indirection.* *(Wheeler).*

# Authenticating to the Server



(SRC-node **as** Excel) **and** bwl may read

SRC **says** WS14 => SRC-node

file foo

*NFS*

*Excel* *pr*

WS14 **as** Excel **and** bwl

*Logged in user* $K_l^{-1}$

WS14 **and** bwl

*Workstation* $K_{ws}^{-1}$

WS14

bwl $K_{bwl}^{-1}$

*Server*

*network channel*

$K_{ca}$ **says** $K_{bwl}$ => bwl

$K_{ca}$ **says** $K_{ws}$ =>WS14

# Access Control

**Checking access:**

| | | |
|---|---|---|
| Given | a request | *Q* **says** `read` *`O`* |
| | an ACL | *P* may `read` *`O`* |

Check that *Q* speaks for *P* $\boxed{Q => P}$

**Auditing**

Each step is justified by

a signed statement, or

a rule

# Authenticating a Channel

**Authentication —** who can send on a channel.

$C$ **=>** $P$; $C$ is the channel, $P$ the sender.

**To get new** $C$ => $P$ **facts**, must trust some principal, a *certification authority*, to tell them to you.

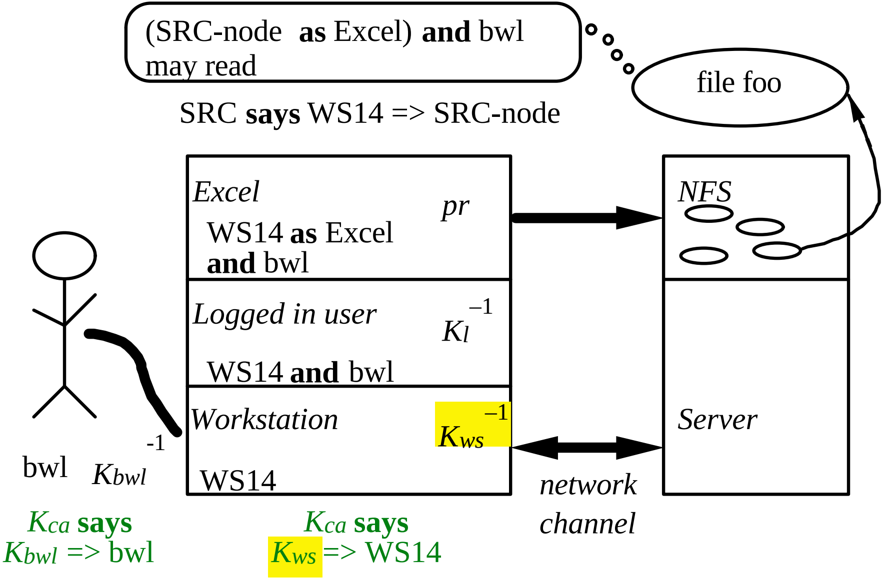
Simplest: trust $K_{ca}$ to authenticate any name:

$\boxed{K_{ca} \Rightarrow \text{Anybody}}$

**Then** *CA* **can authenticate channels:**

$K_{ca}$ **says** $K_{ws}$ => `WS`
$K_{ca}$ **says** $K_{bwl}$ => `bwl`

# Authenticated Channels: Example



(SRC-node **as** Excel) **and** bwl may read

file foo

SRC **says** WS14 => SRC-node

*Excel* *pr*

WS14 **as** Excel **and** bwl

*Logged in user* $K_l^{-1}$

WS14 **and** bwl

*Workstation* $K_{ws}^{-1}$

WS14

*NFS*

*Server*

*network channel*

bwl $K_{bwl}^{-1}$

$K_{ca}$ **says** $K_{bwl}$ => bwl

$K_{ca}$ **says** $K_{ws}$ => WS14

# Groups and Group Credentials

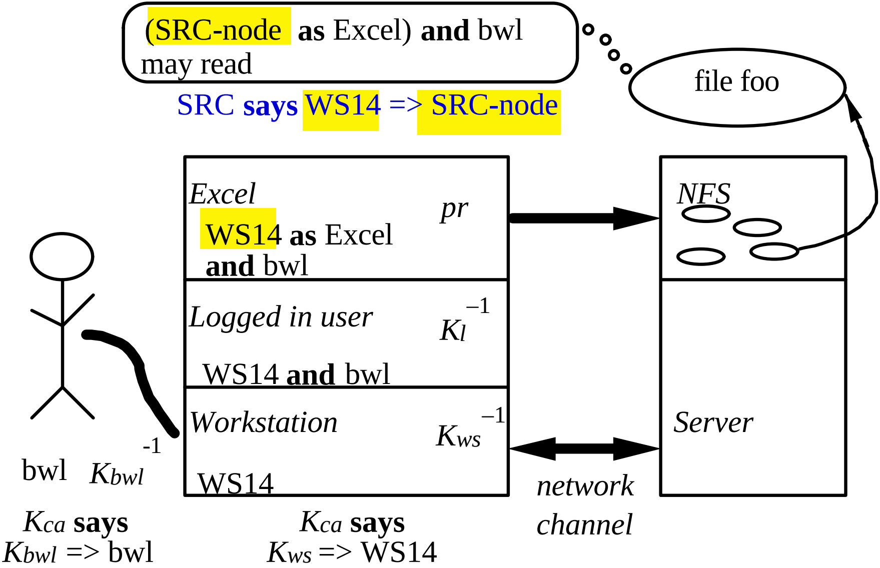
**Defining groups:** A group is a principal; its members speak for it.

Lampson => SRC
Taylor => SRC
. . .

**Proving group membership:** Use certificates.

$K_{src}$ **says** `Lampson => SRC`
$K_{ca}$ **says** $K_{src}$ `=> SRC`

# Authenticating a Group



(SRC-node **as** Excel) **and** bwl may read

SRC **says** WS14 => SRC-node

file foo

*Excel* *pr*

WS14 **as** Excel **and** bwl

*Logged in user* $K_l^{-1}$

WS14 **and** bwl

*Workstation* $K_{ws}^{-1}$

WS14

*NFS*

*Server*

*network channel*

bwl $K_{bwl}^{-1}$

$K_{ca}$ **says** $K_{bwl} \Rightarrow$ bwl

$K_{ca}$ **says** $K_{ws} \Rightarrow$ WS14

# Security: Summary

**Ideas**

- Principals
- Channels as principals
- “Speaks for” relation
- Handoff of authority

**Give precise rules.**

**Apply them to cover many cases.**

# References

| | |
|---|---|
| *Hints* | Lampson, Hints for Computer System Design. *IEEE Software*, Jan. 1984. |
| *Specifications* | Lamport, A simple approach to specifying concurrent systems. *Communications of the ACM*, Jan. 1989. |
| *Reliable messages* | in Mullender, ed., *Distributed Systems*, Addison-Wesley, 1993 (summer) |
| *Transactions* | Gray and Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993. |
| *Security* | Lampson, Abadi, Burrows, and Wobber, Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, Nov. 1992. |

# Collaborators

| | |
|---|---|
| Charles Simonyi | Bravo: WYSIWYG editor |
| Bob Sproull | Alto operating system<br>Dover: laser printer<br>Interpress: page description language |
| Mel Pirtle | 940 project, Berkeley Computer Corp. |
| Peter Deutsch | 940 operating system<br>QSPL: system programming language |
| Chuck Geschke<br>Jim Mitchell<br>Ed Satterthwaite | Mesa: system programming language |
| Jim Horning | Euclid: verifiable programming language |
| Ron Rider<br>Gary Starkweather | Ears: laser printer |
| Severo Ornstein | Dover: laser printer |

# Collaborators

| | |
|---|---|
| Roy Levin | Wildflower: Star workstation prototype<br>Vesta: software configuration |
| Andrew Birrell, Roger Needham, Mike Schroeder | Global name service and authentication |
| Eric Schmidt | System models: software configuration |
| Rod Burstall | Pebble: polymorphic typed language |