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 Networks: AN1, AN2
Charles Simonyi	Bravo WYSIWYG editor
Nancy Lynch	Reliable messages
Howard Sturgis	Transactions
Martin Abadi **Mike Burrows** **Morrie Gasser** **Andy Goldstein** **Charlie Kaufman** **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

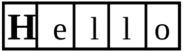
Examples of systems with state:

Data abstractions
Concurrent systems
Distributed systems

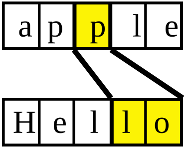
You can't observe the actual state of the system from outside.
All you can see is the results of actions.

Editable Formatted Text

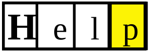
State *text*: sequence of (Char, Property)



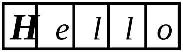
Actions *get(*2) returns ('e', (Times-Roman, ...))

*replace(*3, 5, 2, 3, 

)



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.

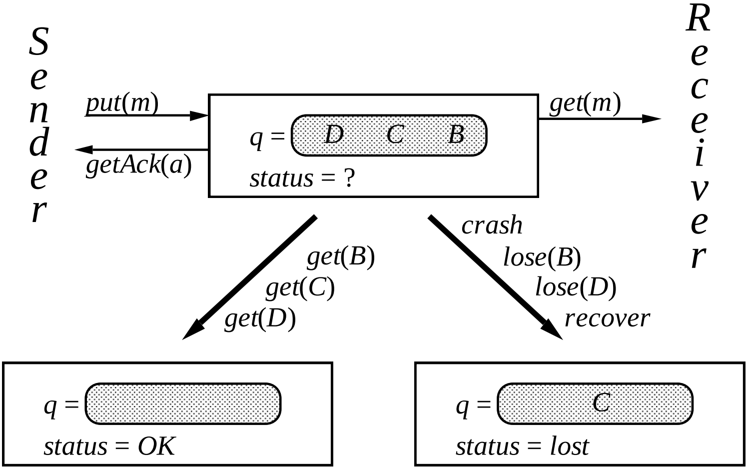
Invent a suitable vocabulary.

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



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, $status$:= ?	*$get(m)$	m first on q	remove head of q, if q = <>, $status$ = ? then $status$:= OK
*$getAck(a)$	$status = a$	$status$:= $lost$			
$lose$	rec_s or rec_r	delete some element from q; if it's the last then $status$:= $lost$, 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.

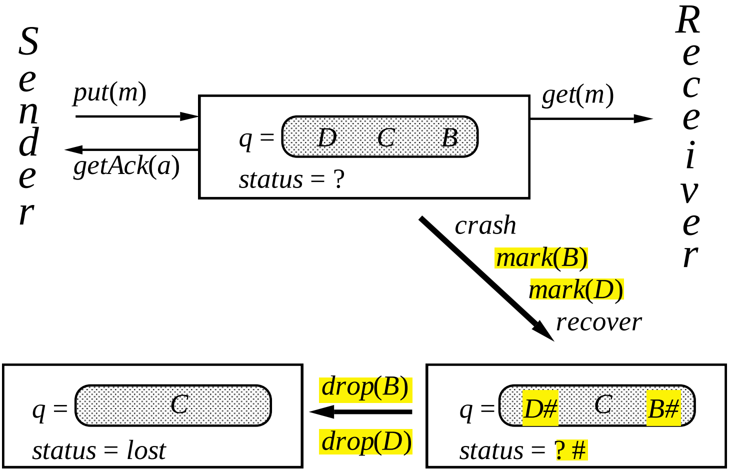
Show that Y *simulates* X:

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



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

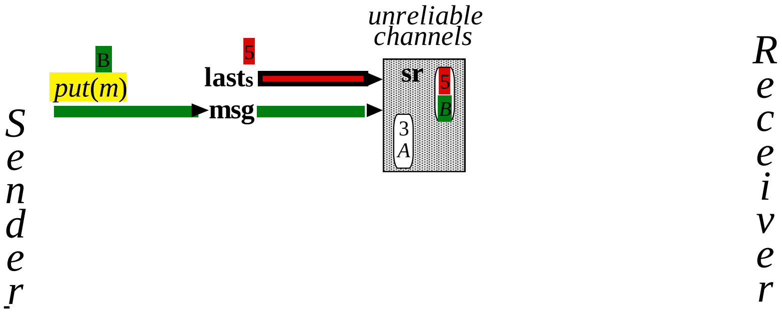
A Generic Protocol G (1)

Sender

actions **state**

Receiver

state **actions**

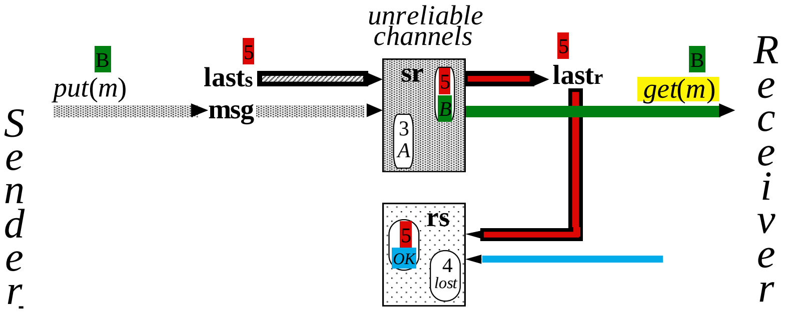


A Generic Protocol G (2)

-

Sender **Receiver**

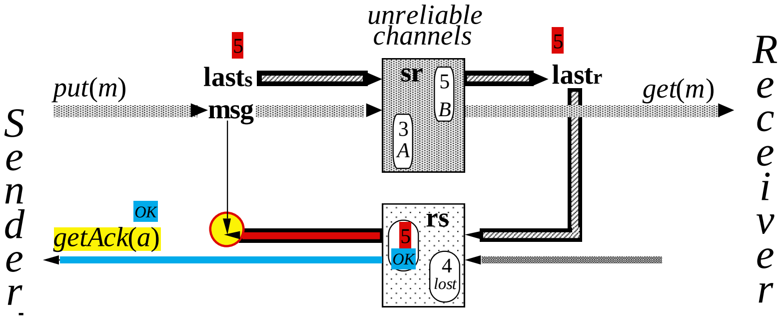
actions **state** **state** **actions**



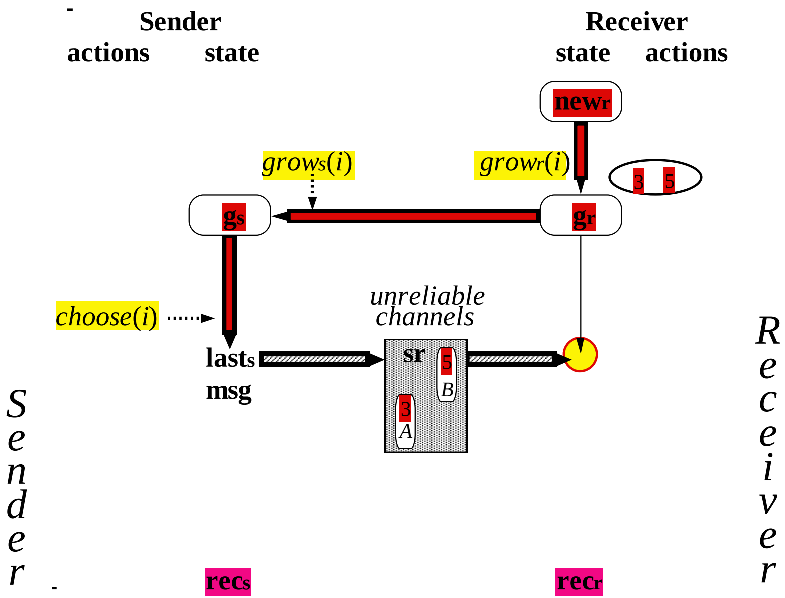
A Generic Protocol G (3)

-

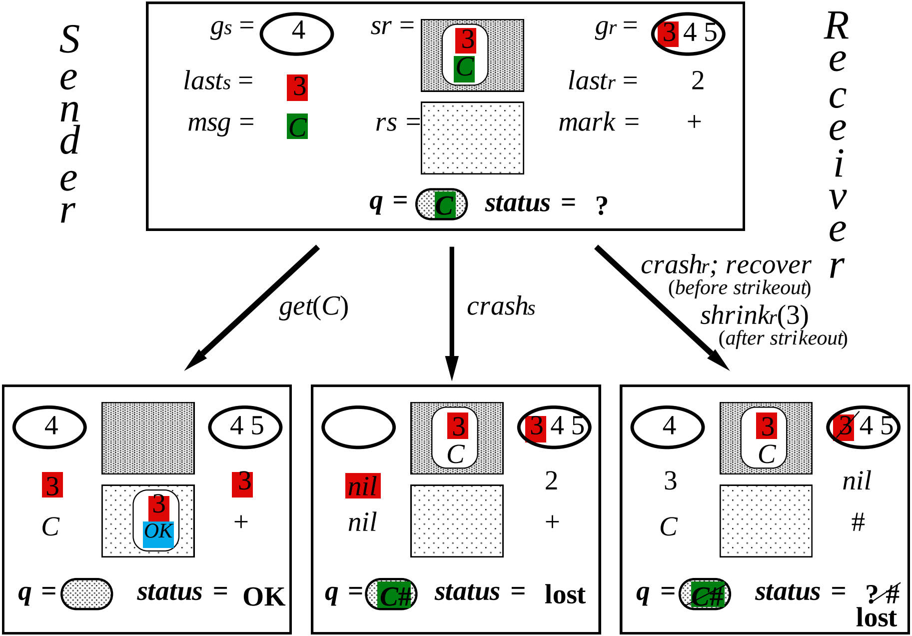
Sender			Receiver	
actions	state		state	actions



A Generic Protocol G (4)



G at Work



Sender
Receiver
gs = 4
sr = 3 C
gr = 3 4 5
lasts = 3
lastr = 2
msg = C
rs =
mark = +
q = C
status = ?
get(C)
crashs
crashr; recover
(before strikeout)
shrinkr(3)
(after strikeout)
4
4 5
3
3
C
3
OK
+
q =
status = OK
3
C
3 4 5
nil
2
nil
+
q = C#
status = lost
4
3
C
3 4 5
3
nil
C
#
q = C#
status = ? #
lost

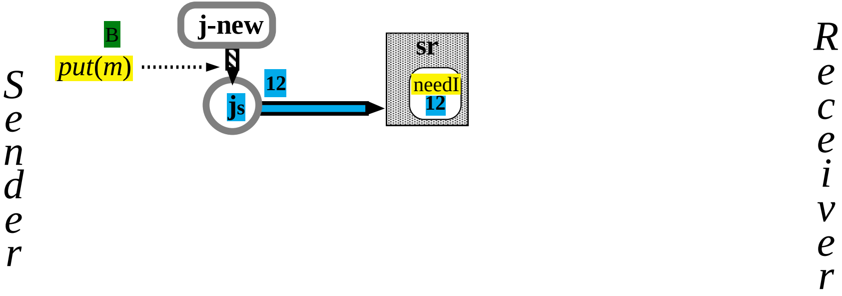
Abstraction Function for G

cur-q	= $\langle msg \rangle$	*if* $msg \neq nil$ and ($last_s = nil$ or $last_s \in g_r$)
	$\langle\ \rangle$	otherwise
old-q	= the messages in *sr* with *i*'s that are good and not = $last_s$	

q	*old-q* + *cur-q*	
status	?	if $cur\text{-}q \neq \langle\ \rangle$
	OK	if $last_s = last_r \neq nil$
	lost	if $last_s \notin (g_r \cup \{last_r\})$ or $last_s = nil$
$rec_{s/r}$	$rec_{s/r}$	

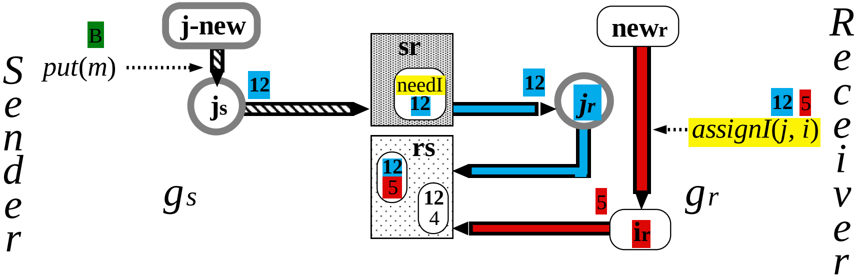
The Handshake Protocol H (1)

-

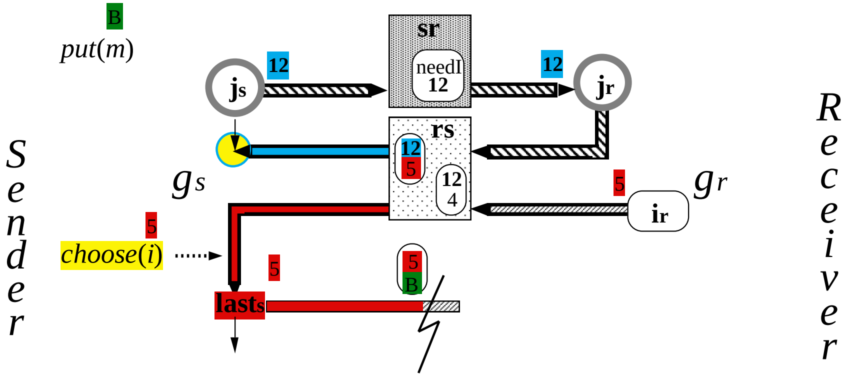


-

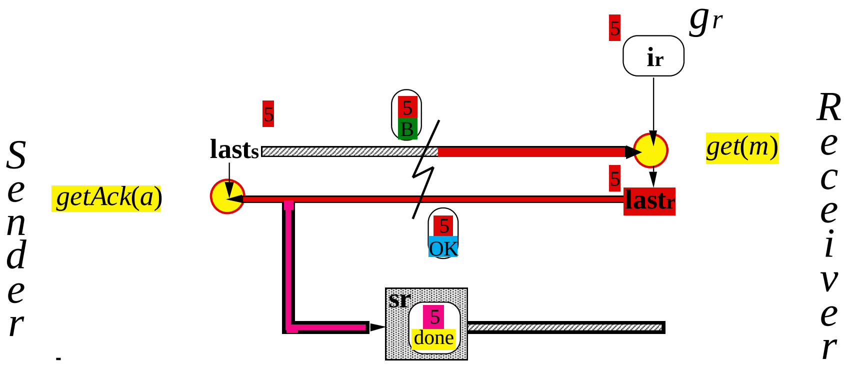
The Handshake Protocol H (2)



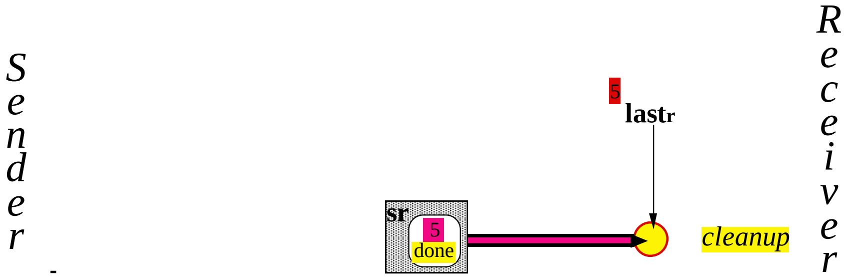
The Handshake Protocol H (3)



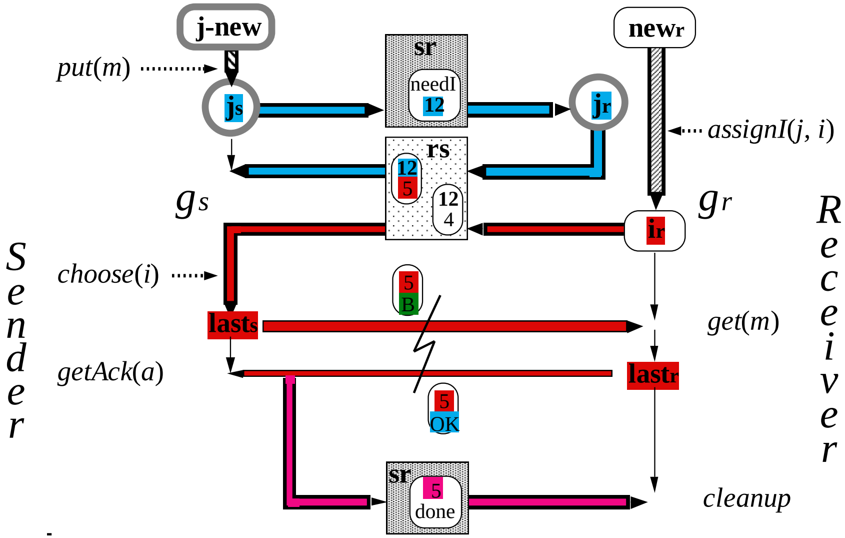
The Handshake Protocol H (4)



The Handshake Protocol H (5)



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 : $State$

X	Y
5	5
$do(x := x-1)$	
4	5
$do(y := y+1)$	
4	6

Name	Guard	Effect
$do(a){:}Val$		$(S, val) := a(S)$

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*

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

Name	Guard	Effect
$do(a){:}Val$		$(s, val) := a(s)$

commit	$S := s$
crash	$s := S$

Server Failures

S , s : *State*

ϕ : {nil, run} := nil

X	Y	x	y	ϕ
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	

Name	Guard	Effect
begin	ϕ = nil	ϕ := run

$do(a)$:Val	ϕ = run	$(s, val) := a(s)$
$commit$	ϕ = run	$S := s$, $\phi :=$ nil
$crash$		$s := S$, $\phi :=$ nil

Note that we clean up the auxiliary state ϕ.

Incremental State Changes: Logs (1)

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

$S = S + L$

$s, \phi = \text{s}, \phi$

X	Y	x	y	*Logs*	ϕ
5	5	5	5		nil
begin; *do*(x:=x–1); *do*(y:=y+1)					
5	5	4	6	x := 4* y := 6*	run
commit					
5	5	4	6	x := 4* y := 6*	nil
crash before *commit*					
5	5	5	5		nil

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

Incremental State Changes: Logs (2)

$S\ ,\ s$: *State*

$L\ ,\ l$: SEQ *Action*

ϕ : {nil, run}

$S = S + L$

$s, \phi = \text{s}, \phi$

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

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

Incremental Log Changes

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

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

X	*Y*	*x*	*y*	*Logs*	Φ	ϕ
5	5	4	6	x := 4* y := 6*	nil	run
flush; *commit*						
5	5	"		x := 4* y := 6*	com	com
apply(x := 4); *apply*(y := 6)						
4	6	"		x := 4 y := 6	com	com
cleanLog; *cleanup*						
4	6	"			nil	nil
crash after *flush*						
4	5	"		x := 4* y := 6*	nil	nil

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

Distributed State and Log

S_i , s_i : *State*
L_i , l_i : SEQ *Action*
Φ_i , ϕ_i : {nil, run*, commit}
S, L, Φ are the products of the S_i, L_i, Φ_i

ϕ = run if all ϕ_i = run
com if any ϕ_i = com
and any $L_i \neq$ < >
nil otherwise

Name	Guard	Effect
begin and *do* as before		
$flush_i$	ϕ_i = run	copy some of l_i to L_i
$prepare_i$	ϕ_i = run, L_i=l_i	Φ_i := run
commit	ϕ = run, L = l	some Φ_i :=ϕ_i :=commit
cleanLog and *cleanup* as before		
$crash_i$		l_i := < >if Φ_i = nil else L_i; s_i := S_i + l_i, ϕ_i := Φ_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.

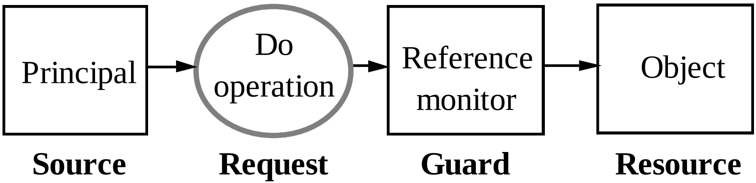
Fewer actions are better. *Less is more.*

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.

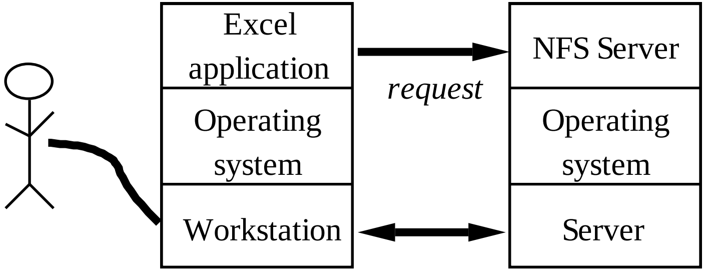


Rules control the operations allowed
for each principal and object.

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: $\boxed{\text{If } A \textbf{ says } B \Rightarrow A \text{ then } B \Rightarrow A}$

Reasonable if *A* is competent and accessible.

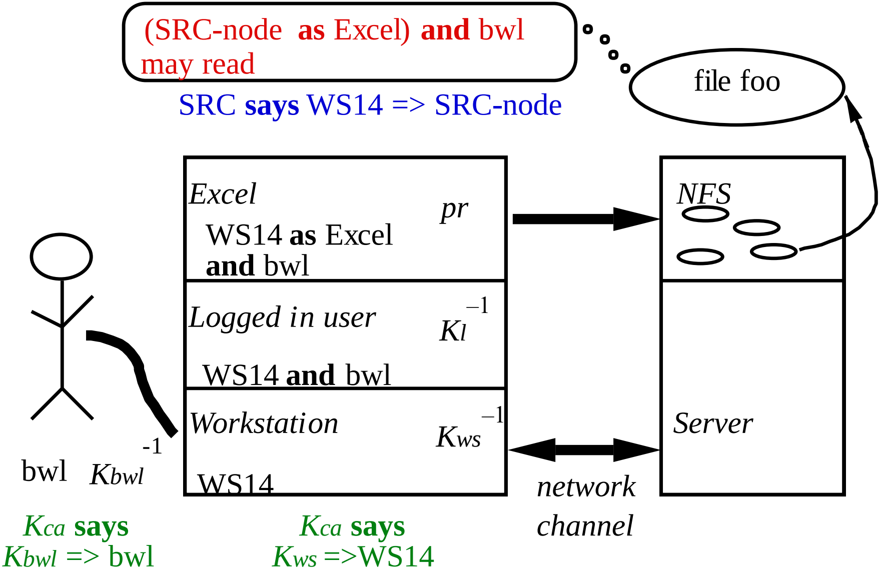
Examples:

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



Access Control

Checking access:

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

Check that Q speaks for P $\boxed{Q \Rightarrow P}$

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.

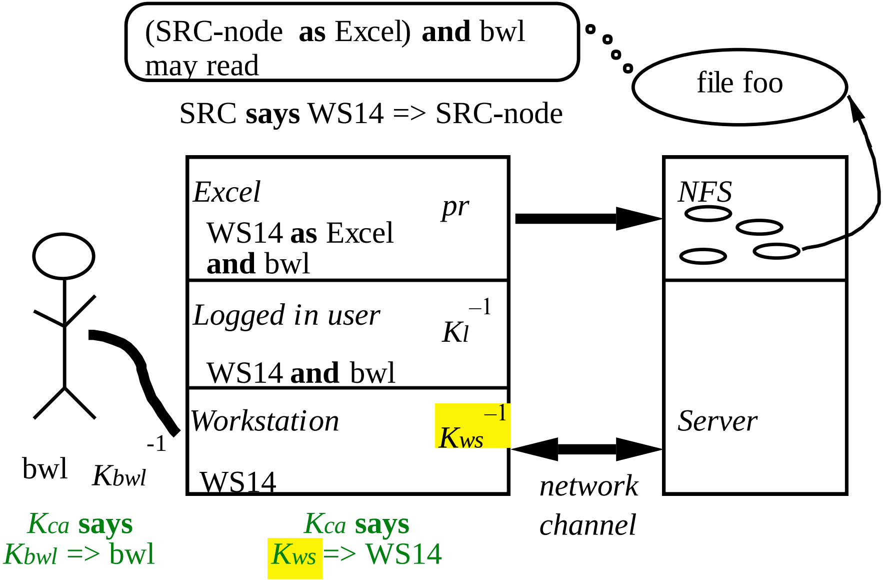
Simplest: trust K_{ca} to authenticate any name:

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

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

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

Authenticated Channels: Example



Groups and Group Credentials

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

Lampson=> SRC

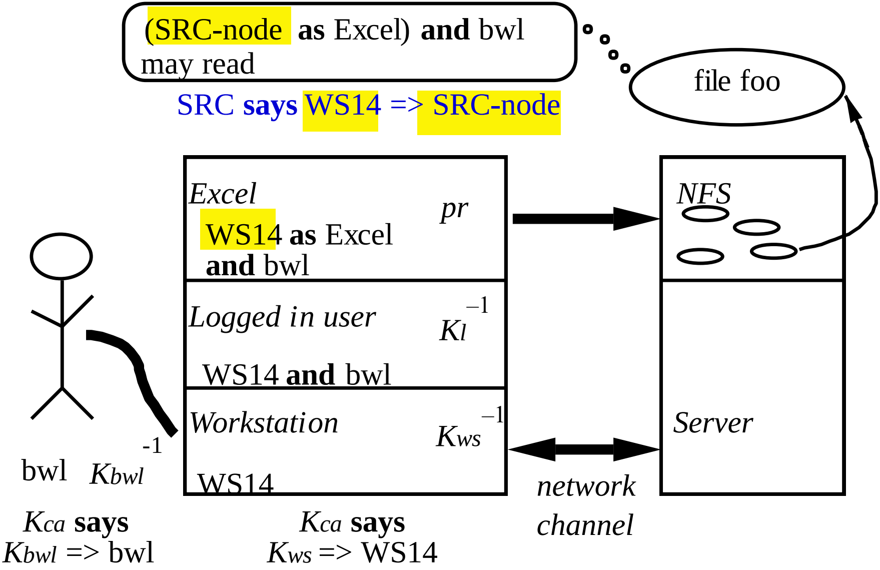
Taylor => SRC

. . .

Proving group membership: Use certificates.

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

Authenticating a Group



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 Dover: laser printer Interpress: page description language
Mel Pirtle	940 project, Berkeley Computer Corp.
Peter Deutsch	940 operating system QSPL: system programming language
Chuck Geschke Jim Mitchell Ed Satterthwaite	Mesa: system programming language
Jim Horning	Euclid: verifiable programming language
Ron Rider Gary Starkweather	Ears: laser printer
Severo Ornstein	Dover: laser printer

Collaborators

Roy Levin	Wildflower: Star workstation prototype 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