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  Security

Mike Burrows

Morrie Gasser

Andy Goldstein

Charlie Kaufman

Ted Wobber
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) apple ) Help

Hello

Hello

Hello

look(0, 5, italic := true)

Hello

Hello

Hello

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

\[ q = \begin{array}{c} D \quad C \quad B \\ \text{status} = ? \end{array} \]

\[ q = \begin{array}{c} D \quad C \quad B \\ \text{status} = OK \end{array} \]

\[ q = \begin{array}{c} D \quad C \quad B \\ \text{status} = lost \end{array} \]
### Spec for Reliable Messages

- **q**: sequence\([M]\)  
  \[\text{:= } <>\]

- **status**: \(\{\text{OK, lost, ?}\}\)  
  \[\text{:= lost}\]

- **rec\(_{s/r}\)**: Boolean  
  \[\text{:= false (short for ‘recovering’)}\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
</table>
| **put\(_m\)** | append \(m\) to \(q\),
  \(\text{status} := ?\) |                                    | *get\(_m\)** | \(m\) first on \(q\),
  if \(q = <>\), \(\text{status} := ?\) | remove head of \(q\),
  then \(\text{status} := OK\) |
| *getAck\(_a\)** | \(\text{status} = a\)  
  \(\text{status} := \text{lost}\) |                                    | lose       | **rec\(_s\)** or
  delete some element from \(q\);  
  \(\text{rec}_r\)**     | if it’s the last then \(\text{status} := \text{lost}\),
  or \(\text{status} := \text{lost}\) |
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.

This always works!
Delayed-Decision Spec: Example

\[ q = \begin{array}{c}
D \\
C \\
B
\end{array} \]

\[ \text{status} = ? \]

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

unreliable channels

Sender
actions
state

Receiver
state
actions

put(m)

msg

lasts

sr

3

A

5

B

5

B
A Generic Protocol G (2)

Sender
actions state

Receiver
state actions

unreliable channels

Sender

Receiver

put(m) lasts msg

sr

lastr

get(m)

rs

OK

lost

B

5

3

A

B

5

5

4

lost

put(m)
A Generic Protocol G (3)

Sender

actions

state

Receiver

state

actions

unreliable
channels

put(m)

lasts

msg

get(m)

Sender

getAck(a)

OK

Receiver

rs

OK

lost

4

5

5

5

3

B

A

sr

lastr

get(m)
A Generic Protocol G (4)

Sender
actions state

Receiver
state actions

sender

receiver

Sender actions state

Receiver state actions

unreliable channels

choose(i)

newr

growr(i)

grows(i)

lasts

msg

recr

recs

 gs

 gr

sr

A

B

S

e
n
d

e
r
sr
3
A

msg
G at Work

Sender

\[ gs = \boxed{4} \]
\[ lasts = 3 \]
\[ msg = C \]
\[ q = C \]
\[ status = ? \]

Receive

\[ sr = \boxed{3} \]
\[ msg = C \]
\[ rs = \boxed{sc} \]
\[ lastr = 2 \]
\[ mark = + \]

\[ gr = \boxed{345} \]

\[ msg = C \]
\[ sr = sc \]
\[ lastr = 2 \]
\[ mark = + \]

\[ qs = C\# \]
\[ status = ? \]

\[ q = C\# \]
\[ status = lost \]

\[ q = C\# \]
\[ status = lost \]
Abstraction Function for G

\[ cur-q = \begin{cases} <\text{msg}> & \text{if } \text{msg} \neq \text{nil} \text{ and } (\text{last}_s = \text{nil} \text{ or } \text{last}_s \in \text{gr}) \\ <> & \text{otherwise} \end{cases} \]

\[ old-q = \text{the messages in } sr \text{ with } i's \text{ that are good and not } = \text{last}_s \]

\[ q = old-q + cur-q \]

\[ \text{status} \]

\[ \begin{align*}
? & \quad \text{if } cur-q \neq <> \\
OK & \quad \text{if } last_s = last_r \neq \text{nil} \\
lost & \quad \text{if } last_s \not\in (gr \cup \{last_r\}) \text{ or } last_s = \text{nil}
\end{align*} \]

\[ rec_{s/r} \]

\[ rec_{s/r} \]
The Handshake Protocol H (1)

Sender

Receiver

put(m)

j-new

S

R

sender

receiver

B

12

j

sr

needI

12
The Handshake Protocol H (2)

Sender

Receiver

j-new

put(m)

js

jr

sr

needI

12

rs

12

4

newr

12

assignI(j, i)

B

S

gs

12

5

gr

12

5

B

12

12

12

5

12

5

12

4

12

5
The Handshake Protocol H (3)

Sender

\(choose(i)\)

\(put(m)\)

Receiver

\(B\)

\(S\)

\(E\)

\(n\)

\(d\)

\(e\)

\(r\)

\(g_s\)

\(last_s\)

\(r_s\)

\(n\)

\(e\)

\(d\)

\(l\)

\(s\)

\(12\)

\(5\)

\(12\)

\(4\)

\(12\)

\(12\)

\(5\)

\(B\)

\(i_r\)

\(g_r\)
The Handshake Protocol H (4)
The Handshake Protocol H (5)
The Handshake Protocol H (6)

\[
\begin{align*}
\text{j-new} & \quad \text{put}(m) \rightarrow \text{js} \\
\text{sr} & \quad \text{choose}(i) \rightarrow \text{jr} \\
\text{rs} & \quad \text{get}(m) \rightarrow \text{lasts} \\
\text{sr} & \quad \text{getAck}(a) \rightarrow \text{lastr} \\
\end{align*}
\]

\[
\begin{align*}
\text{assignI}(j, i) & \quad \text{newr} \\
\end{align*}
\]

Sender:
- choose(i)
- put(m)
- getAck(a)
- done

Receiver:
- assignI(j, i)
- get(m)
- cleanup

\[
\begin{align*}
\text{lasts} & \quad \text{gs} \\
\text{lastr} & \quad \text{gr} \\
\end{align*}
\]
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 ⊆ 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

\[ S : \text{State} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>do(a):\text{Val}</td>
<td></td>
<td>((S, \text{val}) := a(S))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- \(a(S)\): State
- \(\text{do} (x := x - 1)\)
- \(\text{do} (y := y + 1)\)
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

\[ S, s : \text{State} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>do(a):Val</td>
<td></td>
<td>((s, val) := a(s))</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c|c|c|c|c}
X & Y & x & y \\
\hline
5 & 5 & 5 & 5 \\
5 & 5 & 4 & 6 \\
5 & 5 & 5 & 5 \\
\end{array}
\]

\[\text{commit}\]

\[\text{crash before commit}\]
commit

\[ S := s \]

crash

\[ s \; := \; S \]
Server Failures

\[ S, s : \text{State} \]

\[ \phi : \{\text{nil, run}\} := \text{nil} \]

\begin{tabular}{|c|c|} \hline \( X \) & \( Y \) \\ \hline \end{tabular}

\begin{tabular}{|c|c|} \hline \( x \) & \( y \) \\ \hline \end{tabular}

\begin{tabular}{|c|} \hline \( \phi \) \\ \hline \end{tabular}

\begin{tabular}{|c|c|c|c|} \hline 5 & 5 & 5 & 5 \\ \hline 5 & 5 & 4 & 6 \\ \hline 4 & 6 & 4 & 6 \\ \hline \end{tabular}

\begin{tabular}{|c|} \hline \( \text{commit} \) \\ \hline \end{tabular}

\begin{tabular}{|c|} \hline \( \text{crash before commit} \) \\ \hline \end{tabular}

\begin{tabular}{|c|} \hline 5 & 5 & 5 & 5 \\ \hline \end{tabular}

\begin{tabular}{|c|c|c|} \hline \( \text{Name} \) & \( \text{Guard} \) & \( \text{Effect} \) \\ \hline \text{begin} & \( \phi = \text{nil} \) & \( \phi := \text{run} \) \\ \hline \end{tabular}
\[do(a): Va \quad \phi = \text{run} \quad (s, \text{val}) := a(s)\]

\[
\begin{align*}
\text{commit} & \quad \phi = \text{run} \quad S := s, \ \phi := \text{nil} \\
\text{crash} & \quad s := S, \ \phi := \text{nil}
\end{align*}
\]

Note that we clean up the auxiliary state \(\phi\).
Incremental State Changes: Logs (1)

\[ S, s : \text{State} \]
\[ L, l : \text{SEQ Action} \quad := \langle \rangle \]
\[ \phi : \{\text{nil, run}\} \quad := \text{nil} \]

\[ S = S + L \]
\[ s, \phi = s, \phi \]

```
<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>x</th>
<th>y</th>
<th>Logs</th>
<th>(\phi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>nil</td>
<td></td>
</tr>
</tbody>
</table>

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*
```

```
crash before commit
```

```
5 5 5 5
```

\(\phi\)
<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>$\phi = \text{nil}$</td>
<td>$\phi := \text{run}$</td>
</tr>
<tr>
<td>$do(a):Val$</td>
<td>$\phi = \text{run}$</td>
<td>$(s, \text{val}) := a(s), l +:= a$</td>
</tr>
<tr>
<td>commit</td>
<td>$\phi = \text{run}$</td>
<td>$L := l, \phi := \text{nil}$</td>
</tr>
<tr>
<td>crash</td>
<td></td>
<td>$l := L, s := S+L, \phi := \text{nil}$</td>
</tr>
</tbody>
</table>
Incremental State Changes: Logs (2)

\[ S, s : \text{State} \]
\[ L, l : \text{SEQ Action} \]
\[ \phi : \{\text{nil, run}\} \]

\[ S = S + L \]
\[ s, \phi = s, \phi \]

\[
\begin{array}{cccccccc}
X & Y & x & y & \text{Logs} & \phi \\
5 & 5 & 4 & 6 & x := 4* & \text{nil} \\
 & & & & y := 6* & \\
apply(x := 4) & & & & & \\
4 & 5 & " & x := 4 & \text{nil} \\
 & & & & y := 6* & \\
apply(y := 6) & & & & & \\
4 & 6 & " & x := 4 & \text{nil} \\
 & & & & y := 6* & \\
cleanLog & & & & & \\
4 & 6 & " & \text{nil} & \\
crash after apply(x := 4) & & & & & \\
4 & 5 & " & x := 4* & \text{nil} \\
 & & & & y := 6* & \\
\end{array}
\]
<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>begin, do, and commit as before</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>apply(a)</td>
<td>a = head(l)</td>
<td>$S := S + a$, $l := \text{tail}(l)$</td>
</tr>
<tr>
<td>cleanLog</td>
<td>$L \text{ in } S$</td>
<td>$L := &lt;&gt;$</td>
</tr>
<tr>
<td>crash</td>
<td></td>
<td>$l := L$, $s := S + L$, $\phi := \text{nil}$</td>
</tr>
</tbody>
</table>
### Incremental Log Changes

\[ S, s : State \]

\[ L, l : SEQ Action \]

\[ \Phi, \phi : \{\text{nil, run*}, \text{commit}\} \]

\[ L = L \text{ if } \phi = \text{com} \text{ else } <> \]

\[ \phi = \phi \text{ if } \phi \neq \text{com} \text{ else } \text{nil} \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>x</th>
<th>y</th>
<th>Logs</th>
<th>\Phi</th>
<th>\phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>x := 4*</td>
<td>nil</td>
<td>run</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y := 6*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**flush; commit**

| 5 | 5 |   |   | x := 4*  | com | com |
|   |   |   |   | y := 6*  |     |     |

**apply(x := 4); apply(y := 6)**

| 4 | 6 |   |   | x := 4   | com | com |
|   |   |   |   | y := 6   |     |     |

**cleanLog; cleanup**

| 4 | 6 |   |   |   | nil | nil |

**crash after flush**

<p>| 4 | 5 |   |   | x := 4* | nil | nil |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin and do as before</td>
<td></td>
<td></td>
</tr>
<tr>
<td>flush</td>
<td>( \phi = \text{run} )</td>
<td>copy some of ( l ) to ( L )</td>
</tr>
<tr>
<td>commit</td>
<td>( \phi = \text{run}, ; L = l )</td>
<td>( \Phi := \phi := \text{commit} )</td>
</tr>
<tr>
<td>apply((a))</td>
<td>( \phi = \text{commit}, &quot;)</td>
<td>&quot;</td>
</tr>
<tr>
<td>cleanLog</td>
<td>head((L)) in ( S )</td>
<td>( L := \text{tail}(L) )</td>
</tr>
<tr>
<td></td>
<td>or ( \phi = \text{nil} )</td>
<td></td>
</tr>
<tr>
<td>cleanup</td>
<td>( L = &lt;&gt; )</td>
<td>( \Phi := \phi := \text{nil} )</td>
</tr>
<tr>
<td>crash</td>
<td>( l := &lt;&gt; ) if ( \Phi = \text{nil} ) else ( L );</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( s := S + l, ; \phi := \Phi )</td>
<td></td>
</tr>
</tbody>
</table>
Distributed State and Log

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

\[ \phi = \text{run if all } \phi_i = \text{run} \]
\[ \text{com if any } \phi_i = \text{com and any } L_i \neq <> \]
\[ \text{nil otherwise} \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Guard</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin and do as before</td>
<td></td>
<td></td>
</tr>
<tr>
<td>flush_i</td>
<td>( \phi_i = \text{run} )</td>
<td>copy some of ( l_i ) to ( L_i )</td>
</tr>
<tr>
<td>prepare_i</td>
<td>( \phi_i = \text{run}, L_i = l_i )</td>
<td>( \Phi_i := \text{run} )</td>
</tr>
<tr>
<td>commit</td>
<td>( \phi = \text{run}, L = l )</td>
<td>some ( \Phi_i := \phi_i := \text{commit} )</td>
</tr>
<tr>
<td>cleanLog and cleanup as before</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crash_i</td>
<td>( l_i := &lt;&gt; ) if ( \Phi_i = \text{nil} ) else ( L_i );</td>
<td>( s_i := S_i + l_i, \phi_i := \Phi_i )</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Principal</th>
<th>Operation</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taylor</td>
<td>Read</td>
<td>File “Raises”</td>
</tr>
<tr>
<td>Jones</td>
<td>Pay invoice 4325</td>
<td>Account Q34</td>
</tr>
</tbody>
</table>
Schwarzkopf  Fire three rounds  Bow gun
A Distributed System

- Excel application
- Operating system
- Workstation

→ request

- NFS Server
- Operating system
- Server
# Principals

<table>
<thead>
<tr>
<th><strong>Authentication:</strong></th>
<th>Who sent a message?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authorization:</strong></td>
<td>Who is trusted?</td>
</tr>
</tbody>
</table>

Principal — abstraction of "who":

<table>
<thead>
<tr>
<th>People</th>
<th>Lampson, Taylor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machines</td>
<td>VaxSN12648, Jumbo</td>
</tr>
<tr>
<td>Services</td>
<td>SRC-NFS, X-server</td>
</tr>
<tr>
<td>Groups</td>
<td>SRC, DEC-Employees</td>
</tr>
<tr>
<td>Channels</td>
<td>Key #7438</td>
</tr>
</tbody>
</table>
Theory of Principals

Principal says statement

Lampson says “read /SRC/Lampson/foo”
SRC-CA says “Lampson’s key is #7438”

Principal $A$ speaks for $B$

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

A secure channel:
says things directly

If $P$ is the only sender on $C$

Examples

Lampson $\Rightarrow$ SRC
Key #7438 $\Rightarrow$ Lampson
Handing Off Authority

Handoff rule: If \( A \) says \( B \Rightarrow A \) then \( B \Rightarrow A \)

Reasonable if \( A \) is competent and accessible.

Examples:

\( \text{SRC says Lampson} \Rightarrow \text{SRC} \)

\( \text{Node key says Channel key} \Rightarrow \text{Node key} \)

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

(SRC-node \textit{as Excel}) \textbf{and} bw1 may read

\textbf{SRC says WS14 $\Rightarrow$ SRC-node}

\begin{itemize}
  \item Excel
    \begin{itemize}
      \item WS14 \textit{as Excel}
      \item \textbf{and} bw1
    \end{itemize}
  \item Logged in user
    \begin{itemize}
      \item WS14 \textbf{and} bw1
    \end{itemize}
  \item Workstation
    \begin{itemize}
      \item WS14
    \end{itemize}
\end{itemize}

\textbf{Kca says} $K_{bw1} \Rightarrow bw1$

\textbf{Kca says} $K_{ws} \Rightarrow WS14$

\textbf{file foo}

\textbf{network channel}

\textbf{Server}

\textbf{NFS}

\textbf{Excel}

\textbf{Logged in user}

\textbf{Workstation}

\textbf{bwl}

$K_{bw1}$

$K_{ws}$
Access Control

Checking access:

Given a request $Q$ says read $O$

an ACL $P$ may read $O$

Check that $Q$ speaks for $P$ $Q \Rightarrow P$

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:

\[ K_{ca} => \text{Anybody} \]

Then \( CA \) can authenticate channels:

\[ K_{ca} \text{ says } K_{ws} => WS \]
\[ K_{ca} \text{ says } K_{bw} => bw \]
Authenticated Channels: Example

(SRC-node as Excel) and bwl may read

 SRC says WS14 => SRC-node

Excel
WS14 as Excel
and bwl
Logged in user
WS14 and bwl
Workstation
WS14

K KL
ws => WS14
SRC says WS14 => SRC-node
WS14
K bwl => bwl
Kca says
Kbw => bwl

file foo

NFS

pr

network channel

Server

Kca says
Kbwl => bwl

K ca says
K bw => bwl

Kca says
K ws => WS14
Groups and Group Credentials

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

Lampson $\Rightarrow$ SRC
Taylor $\Rightarrow$ SRC

\[ \ldots \]

Proving group membership: Use certificates.

$K_{src}$ says Lampson $\Rightarrow$ SRC
$K_{ca}$ says $K_{src} \Rightarrow$ SRC
AUTHENTICATING A GROUP

(SRC-node as Excel) and bwl
may read

SRC says WS14 => SRC-node

Excel

pr

WS14 as Excel

and bwl

Logged in user

-1

Kl

WS14 and bwl

Workstation

-1

Kws

WS14

Kca says

Kbw1 => bwl

bwl

Kbw1

Kca says

Kbw1 => bwl

bwl

Kca says

Kws => WS14

WS14

NFS

Server

file foo

network channel
Security: Summary

Ideas

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

Give precise rules.

Apply them to cover many cases.
## References

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable messages</td>
<td>in Mullender, ed., <em>Distributed Systems</em>, Addison-Wesley, 1993 (summer)</td>
</tr>
<tr>
<td>Security</td>
<td>Lampson, Abadi, Burrows, and Wobber, Authentication in distributed systems: Theory and</td>
</tr>
</tbody>
</table>
## Collaborators

<table>
<thead>
<tr>
<th>Name</th>
<th>Project/Invention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charles Simonyi</td>
<td>Bravo: WYSIWYG editor</td>
</tr>
<tr>
<td>Bob Sproull</td>
<td>Alto operating system</td>
</tr>
<tr>
<td></td>
<td>Dover: laser printer</td>
</tr>
<tr>
<td></td>
<td>Interpress: page description language</td>
</tr>
<tr>
<td>Mel Pirtle</td>
<td>940 project, Berkeley Computer Corp.</td>
</tr>
<tr>
<td>Peter Deutsch</td>
<td>940 operating system</td>
</tr>
<tr>
<td></td>
<td>QSPL: system programming language</td>
</tr>
<tr>
<td>Chuck Geschke</td>
<td>Mesa: system programming language</td>
</tr>
<tr>
<td>Jim Mitchell</td>
<td></td>
</tr>
<tr>
<td>Ed Satterthwaite</td>
<td></td>
</tr>
<tr>
<td>Jim Horning</td>
<td>Euclid: verifiable programming language</td>
</tr>
<tr>
<td>Ron Rider</td>
<td>Ears: laser printer</td>
</tr>
<tr>
<td>Gary Starkweather</td>
<td></td>
</tr>
<tr>
<td>Severo Ornstein</td>
<td>Dover: laser printer</td>
</tr>
</tbody>
</table>
## Collaborators

<table>
<thead>
<tr>
<th>Name</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roy Levin</td>
<td>Wildflower: Star workstation prototype</td>
</tr>
<tr>
<td></td>
<td>Vesta: software configuration</td>
</tr>
<tr>
<td>Andrew Birrell, Roger Needham, Mike Schroeder</td>
<td>Global name service and authentication</td>
</tr>
<tr>
<td>Eric Schmidt</td>
<td>System models: software configuration</td>
</tr>
<tr>
<td>Rod Burstall</td>
<td>Pebble: polymorphic typed language</td>
</tr>
</tbody>
</table>