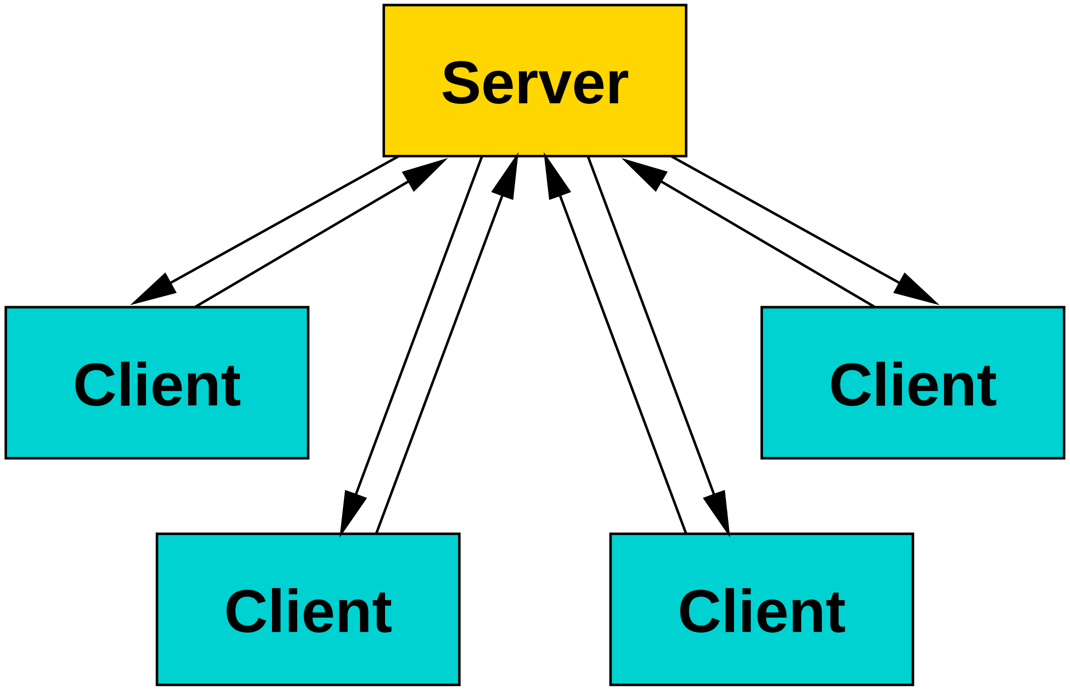
On Engineering a Distributed Algorithm

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Distributed Environment



Problem

Write a server that schedules some resource among C clients such that no more than R clients operate on it at the same time.

```
void Client(Channel server, Channel client)
{
   while (true)
   {
      ...
      send(server, REQUEST);
      grant = receive(client);
      ...
      send(server, RELEASE);
   }
}
```

```
Channel[C] server, client;
parallel
{
   Server(server, client);
   parallel(int i=0; i < C; i++)
      Client(server, client);
}
```

Implement `Server` such that it meets the specification.

Result

Software engineer returns a program.

```
void Server(Channel[C] server, Channel[C] client)
{
  Array<bool> wait = new(C, false);
  int counter = R;
  while (true) {
    int i = select(server);
    if (receive(server[i]) == REQUEST) {
      if (counter > 0) {
        send(client[i], GRANT);
        counter--;
      }
      else
        wait[i] = true;
      }
    }
```

```
    ...
    else {
      int j = exists(wait);
      if (j != -1) {
        send(client[j], GRANT);
        wait[j] = false;
      }
      else
        counter++;
    }
  }
}
```

Confidence in the Solution

Discussion between customer and software engineer.

- C: How does your program solve the problem?
  - SE: (hand-waving explanation refering to some diagrams illustrating example runs).
- C: Why do you believe that the program is correct?
  - SE: I've tested it for various C and R.
- C: So the program runs the same for any parameters.
  - SE: No, different situations may arise for other parameters.
- C: But for a particular choice, you've tested all possibilities.
  - SE: No, there are too many possible program runs; I've just tested some of them.
- C: So, if the program works for these, it will work for all others.
  - SE: Not necessarily; other runs may lead to not yet covered program situations.

Confidence in the Solution

- C: But you can give an argument why your program solves the general problem.
  - SE: (again demonstrates the operation of the program on a particular example run).
- C: At least you've followed a design methodology which is trusted to yield correct solutions for this kind of problems.
  - SE: Actually, I've just written the code from scratch.
- C: So how can I rely on your solution?

Is this engineering?

Engineering Distributed Software

- Technologies.
  - Communication protocols, standards, APIs, ...
  - Class libraries: STL, JDK, JFC, ...
  - Middleware: CORBA, DCOM, RMI, Jini, ...
  - Application Frameworks: San Francisco, EJB, ...
- Mental and Linguistic Model.
  - Formulating the problem.
  - Designing a solution.
  - Reasoning about its correctness.

The reliability of (distributed) software depends on engineers who are able to reason and to talk about the artefacts they design.

Modeling Distributed Systems

Need a formal framework for our design.

- Various Approaches.
  - Logic-based: axiomatic systems, UNITY, temporal logic, ...
  - Algebra-based: CSP, CCS, π-calculus, ...
  - ...
- Temporal Logic.
  - System P and specification S described by predicates.
  - P holds on *possible* sequences of system states.
  - S holds on *legal* sequences of system states.
  - System meets specification: $P \Rightarrow S$.

Unambiguous formulation of the problem becomes possible.

Correctness Statement

Write a server that schedules some resource among C clients such that no more than R clients operate on it at the same time.

forall C, R:	*System parameters.*
forall *pc*:	*Client states (program counters).*
System$_{C,R}$(*pc*) $\Rightarrow$	*System.*
always	*Specification.*
and	
$R \geq$ **number** $i \in [1, C]$: $pc_i = \underline{\text{resource}}$	*Safety: no more than R processes use resource.*
forall $i \in [1, C]$: $(pc_i = \underline{\text{request}} \Rightarrow$	*Progress: every requesting process gets resource.*
eventually $pc_i = \underline{\text{resource}})$	

Goal for the design of the system.

System Model

System$_{C,R}$(*pc*) :$\Leftrightarrow$	
var *server*, *client*, ...:	*Local system state.*
and	
Init$_{C,R}$(*pc*, *server*, *client*, ...)	*Condition on initial state of the system.*
always Action$_{C,R}$(*pc*, *server*, *client*, ...)	*Transition between system states.*
Action$_{C,R}$(*pc*, *server*, *client*, ...) :$\Leftrightarrow$	*An action of the system.*
or	
Server$_{C,R}$(*server*, *client*, ...)	*An action of the server.*
exists $i \in [1, C]$:	
Client$_i$(*pc*, *server*, *client*, ...)	*An action of some of the clients.*

Interleaved actions of server and clients on state variables.

Client Model

$\text{Client}_i(pc, server, client)$:⇔
or
(1) $pc_i = \underline{\text{other}} \wedge pc'_i = \underline{\text{request}} \wedge$
$\text{changed}(pc)$
(2) $pc_i = \underline{\text{request}} \wedge pc'_i = \underline{\text{grant}} \wedge$
$send_i(server, \underline{\text{request}}) \wedge \text{changed}(pc, server)$
(3) $pc_i = \underline{\text{grant}} \wedge pc'_i = \underline{\text{resource}} \wedge$
$receive_i(client, \underline{\text{grant}}) \wedge \text{changed}(pc, client)$
(4) $pc_i = \underline{\text{resource}} \wedge pc'_i = \underline{\text{release}} \wedge$
$\text{changed}(pc)$
(5) $pc_i = \underline{\text{release}} \wedge pc'_i = \underline{\text{other}} \wedge$
$send_i(server, \underline{\text{release}}) \wedge \text{changed}(pc, server)$

```
Client(server, client):
  loop
    ...                              other
    send(server, request)            request
    grant := receive(client)         grant
    ...                              resource
    send(server, release)            release
  end
```

$\text{send}_i(c, m)$:⇔
$c' = c[i \mapsto c[i] \parallel \langle m \rangle]$

$\text{receive}_i(c, m)$:⇔
$c = c'[i \mapsto \langle m \rangle \parallel c'[i]]$

$\text{changed}(v_1, \ldots, v_n)$:⇔
(for all program variables v except for $v_1, \ldots, v_n$)
and
$v' = v$
...

Solution Idea

Introduce variable *counter* and maintain invariant:

implies
  and — *System.*
    $\text{Init}_{C,R}$(*pc*, *counter* ...)
    always $\text{Action}_{C,R}$(*pc*, *counter*, ...)
  always $\text{Invariant}_{C,R}$(*pc*, *counter*, ...) — *Stronger specification.*

Chosen appropriately, invariant implies safety property:

$\text{Invariant}_{C,R}$(*pc*, *counter*, ...) :⇔
  and
    counter ≥ 0 — *Counter is non-negative.*
    R =
      counter + — *Counter complements number of*
      number i $\in [1, C]$: pc_i = <u>resource</u> — *processes operating on resource.*

Must check whether clients maintains invariant.

Refining the Invariant

- (4) violates invariant.

$$\begin{aligned}&\mathit{R} =\\ &\quad \mathit{counter} +\\ &\quad \textbf{number}\ i \in [1, C]\colon\ \mathit{pc}_i = \underline{\mathsf{resource}} \lor \mathit{pc}_i = \underline{\mathsf{release}}\end{aligned}$$

- (3) violates invariant.

$$\begin{aligned}&\mathit{R} =\\ &\quad \mathit{counter} +\\ &\quad \textbf{number}\ i \in [1, C]\colon\ \mathit{pc}_i = \underline{\mathsf{resource}} \lor \mathit{pc}_i = \underline{\mathsf{release}} +\\ &\quad \textbf{number}\ i \in [1, C],\ j \in [1, \overline{\mathsf{client}_i}]\colon\ \mathit{client}_i[j] = \underline{\mathsf{grant}}\end{aligned}$$

- (5) violates invariant.

$$\begin{aligned}&\mathit{R} =\\ &\quad \mathit{counter} +\\ &\quad \textbf{number}\ i\colon\ \mathit{pc}_i = \underline{\mathsf{resource}} \lor \mathit{pc}_i = \underline{\mathsf{release}} +\\ &\quad \textbf{number}\ i \in [1, C],\ j \in [1, \overline{\mathsf{client}_i}]\colon\ \mathit{client}_i[j] = \underline{\mathsf{grant}} +\\ &\quad \textbf{number}\ i \in [1, C],\ j \in [1, \overline{\mathsf{server}_i}]\colon\ \mathit{server}_i[j] = \underline{\mathsf{release}}\end{aligned}$$

Refined invariant still implies safety property.

Designing the Server

Server must maintain invariant and imply progress:

Server$_{C,R}$(*server*, *client*, *counter*) :⇔
  exists $i \in [1, C]$:
    or
      receive$_i$(*server*, release) $\wedge$ — *receive a release message and increase counter*
        count' $=$ *count*$+1$ $\wedge$
        changed(*server*, *count*)
      receive$_i$(*server*, request) $\wedge$ *count* > 0 $\wedge$ — *send a grant message and decrease counter*
        send$_i$(*client*, grant) $\wedge$ *count*' $=$ *count*-1 $\wedge$
        changed(*server*, *client*, *count*)
      receive$_i$(*server*, request) $\wedge$ *count* $\not> 0$ $\wedge$ — *what now?*
        ...

Need possibility to defer grant messages.

Designing the Server

Introduce variable *wait* to record deferred requests:

Server$_{C,R}$(*server*, *client*, *counter*, *wait*) :⇔
  exists $i \in [1, C]$:
    or
      ...
      receive$_i$(*server*, request) ∧ *count* $\not>$ 0 ∧
        wait' = *wait*$[i \mapsto$ true$]$ ∧
        changed(*server*, *wait*)

Need to guarantee progress:

Invariant$_{C,R}$(*pc*, *server*, *client*, *counter*, *wait*) :⇔
  and
    ...
    exists $j \in [1, C]$: *wait*$[j]$ ⇒   *Grant only deferred as long as necessary.*
      count = 0

Need to maintain refined invariant.

Designing the Server

Need to be careful on receipt of release messages:

Server$_{C,R}$(*server*, *client*, *counter*, *wait*) :⇔
  exists $i \in [1, C]$:
    or
      receive$_i$(*server*, <u>release</u>) ∧ ¬**exists** $j \in [1, C]$: *wait*[j] ∧
        count' = *count*+1 ∧
        changed(*server*, *count*)
      receive$_i$(*server*, <u>release</u>) ∧ **exists** $j \in [1, C]$: (*wait*[j] ∧
        send$_j$(*client*, <u>grant</u>) ∧ *wait*' = *wait*[j ↦ <u>false</u>]) ∧
        changed(*server*, *client*, *wait*)
      ...

Resulting server maintains invariant.

Server Model

Server(*server*, *client*, *count*, *wait*) :⇔ — *a server action*

exists $i \in [1, C]$:

or

receive$_i$(*server*, request) ∧ *count* > 0 ∧ — *receive request and return grant*
send$_i$(*client*, grant) ∧ *count*' $=$ *count*-1 ∧
changed(*server*, *client*, *count*)

receive$_i$(*server*, request) ∧ *count* $\not> 0$ ∧ — *receive request and defer reply*
wait' $=$ *wait*[*i* ↦ true] ∧
changed(*server*, *wait*)

receive$_i$(*server*, release) ∧ **exists** *j* $\in [1, C]$: (*wait*[*j*] ∧ — *receive release and send deferred grant*
send$_j$(*client*, grant) ∧ *wait*' $=$ *wait*[*j* ↦ true]) ∧
changed(*server*, *client*, *wait*)

receive$_i$(*server*, release) ∧ ¬**exists** *j* $\in [1, C]$: *wait*[*j*] ∧ — *receive release and increase counter*
count' $=$ *count*$+1$ ∧
changed(*server*, *count*)

Result of a systematic design.

System Model

Action$_C$(*pc*, *server*, *client*, *count*, *wait*) :⇔ *one action of the system*
  or
    Server(*server*, *client*, *count*, *wait*)
    exists $i \in [1, C]$:
      Client$_i$(*server*, *client*, *pc*)

Init$_{C,R}$(*pc*, *server*, *client*, *count*, *wait*) :⇔ *the initial condition of the system*
  and
    count = R
    forall $i \in [1, C]$:
      pc_i = <u>other</u> ∧
      $server_i = \langle\rangle \wedge client_i = \langle\rangle \wedge wait[i] =$ false

System$_{C,R}$(*pc*) :⇔ *the distributed system*
  exists *server*, *client*, *count*, *wait*:
    and
      Init$_{C,R}$(*pc*, *server*, *client*, *count*, *wait*)
      always Action$_C$(*pc*, *server*, *client*, *count*, *wait*)

System Invariants

forall C, R:
forall *pc*:
  System$_{C,R}$(*pc*) $\Rightarrow$
    always
      and
        count ≥ 0 — *Counter is non-negative.*
        $R =$
          count $+$
          number $i \in [1,C]$: $pc_i = \underline{\text{critical}} \lor pc_i = \underline{\text{release}}$ $+$
          number $i \in [1,C]$, $j \in [1,\overline{c_i}]$: $client_i[j] = \underline{\text{grant}}$
          number $i \in [1,C]$, $j \in [1,\overline{s_i}]$: $server_i[j] = \underline{\text{release}}$ $+$
        — *Counter complements number of processes using resource plus number of pending release and grant messages.*
        exists $j \in [1,C]$: $wait[j] \Rightarrow$
          count $= 0$
        — *Grants only deferred as long as necessary.*

Key knowledge about the designed system.

Server Program

```
Server(server, client):
  var count := R, wait[.] := false
  loop
    var i := select(server)
    if receive(server[i]) = request then
      if count > 0 then
        send(client[i], grant)
        count := count−1
      else
        wait[i] := true
    else // release received
      if exists j ∈ [1, C]: wait[j] then
        take j ∈ [1, C] such that wait[j]
        send(client[j], grant)
        wait[j] := false
      else
        count := count+1
    end
  end
```

```
Server(server, client, count, wait) :⇔
  exists i ∈ [1, C]:
    or
      receive_i(server, request) ∧ count > 0 ∧
        send_i(client, grant) ∧ count' = count−1 ∧
        changed(server, client, count)
      receive_i(server, request) ∧ count ≯ 0 ∧
        wait' = wait[i ↦ true] ∧
        changed(server, wait)
      receive_i(server, release) ∧ exists j: (wait[j] ∧
        send_j(client, grant) ∧ wait' = wait[j ↦ true]) ∧
        changed(server, client, wait)
      receive_i(server, release) ∧ ¬exists j: wait[j] ∧
        count' = count+1 ∧
        changed(server, count)
```

Reliability of the Solution

- Why should I believe that your program is correct?
  - Formal model of the specification.
  - Designed model system such that some invariants are maintained.
  - Can reason that invariants imply specification.
  - Code is implementation of the model system.
- But you have not verified your allegations in detail.
  - With more efforts (costs), reasoning can be refined.
  - Show that model system indeed maintains invariants.
  - Refine invariants such that they indeed imply specification.
  - Show that implementation preserves properties of model.

Design process increases confidence and supports further analysis.

Engineering Distributed Software

Complexity of distributed software is daunting.

- Less and less software is purely sequential and purely local.
  - Reactive systems.
  - Concurrent systems.
  - Distributed systems.
- Distributed software only based on gut is not reliable.
  - Maybe the solution is wrong.
  - Probably the solution is incomplete.
  - Certainly it cannot be *analyzed* and *discussed*.

Software *engineers* must be able to operate in formal models.