Protocols

Concurrency and Distributed Systems
November 2023

Contents

- Communication
- Coordination
- Consensus

Protocols

A protocol is a set of rules for collaboration.

If each party follows the rules that apply to them, then the collaboration will be successful.

The outcome of a successful collaboration may be:

- communication
- coordination or consensus

Communication

In describing a data protocol, we may use processes to represent:

- the components of an implementation: nodes, clients, servers, senders, or receivers;
- assumptions about the underlying media and/or the occurrence and ordering of events at different nodes;
- the service that the protocol is intended to provide.

Verification

If the processes `Protocol`, `Media`, and `Service` describe the components, the assumptions, and the intended service, respectively, then

```
Service [= (Protocol || Media) \ Internal
```

where `Internal` is the set of all events performed by the components that are not part of the service.

Buffers

A perfect service will be a buffer of some capacity.

A buffer is a process that stores and forwards messages so that

- no messages are lost
- the order of messages is preserved

It should always be ready to accept input (unless it is already full) and ready to provide output (unless it is empty).

some degree of buffering is inevitable

Example

```
OnePlaceBuffer =
  let
    Empty =
      in?x -> Full(x)

    Full(x) =
      out!x -> Empty
  within
    Empty
```

Example

```
Buffer(capacity) =
  let
    State(s) =
      length(s) < capacity & in?x -> State(s^<x>)
      []
      length(s) > 0 & out!head(s) -> State(tail(s))
  within
    State(<>)
```

where `length`, `head`, and `tail` are functions returning the length, the head, and the tail of a sequence

Example

```
datatype Message = data.{0..1} | ack
channel in, out : Message

channel sender, receiver : IO . Message
datatype IO = send | receive

aMedia = {| sender, receiver |}
```

```
MessageMedium =
  let
    Ready =
      sender.send?x -> Hold(x)
    Hold(x) =
      receiver.receive!x -> Ready
  within
    Ready
```

```
AckMedium =
  let
    Ready =
      receiver.send?x -> Hold(x)
    Hold(x) =
      sender.receive!x -> Ready
  within
    Ready
```

```
aSender = {| sender, in |}

Sender =
  let
    Ready =
      in?x -> Send(x)
    Send(x) =
      sender.send!x -> Wait
    Wait =
      sender.receive?a -> Ready
  within
    Ready
```

```
aReceiver = {| receiver, out |}

Receiver =
  let
    Ready =
      receiver.receive?x -> Output(x)
    Output(x) =
      out!x -> Acknowledge(x)
    Acknowledge(x) =
      receiver.send!ack -> Ready
  within
    Ready
```

see: alternative

```
System =
  ( (Sender ||| Receiver)
    [| aMedia |]
    (MessageMedia ||| AckMedia))
       \ aMedia

assert Buffer(1) [FD= System
```

This check succeeds. Would it succeed with the definition of `Receiver` on the next slide?

```
aReceiver = {| receiver, out |}

Receiver =
  let
    Ready =
      receiver.receive?x -> Acknowledge(x)
    Acknowledge(x) =
      receiver.send!ack -> Output(x)
    Output(x) =
      out!x -> Ready
  within
    Ready
```

see: original

Lossy Media

```
MessageLossyMedia =
  let
    Pass =
      sender.send?x ->
        receiver.receive!x ->
          (Drop |~| Pass)
    Drop =
      sender.send?x ->
        Pass
  within
    (Drop |~| Pass)
```

Retransmit

```
channel timeout

Sender =
  let
    Ready =
      in?x -> Send(x)
    Send(x) =
      sender.send!x -> Wait(x)
    Wait(x) =
      sender.receive?a -> Ready
      []
      timeout -> Send(x)
within
  Ready
```

```
System =
  ((Sender ||| Receiver)
  [| aMedia |]
  (MessageLossyMedia ||| AckMedia) \ aMedia

Service = Buffer(1)

assert Service [FD= System
```

Sliding windows

The sender process could send (and retain) more than one message pending acknowledgement.

```
-- modulus (largest value for message numbering)
msn = 10

-- extract message or number from data pair
message(x.n) = x
number(x.n) = n
```

```
Sender(N) =
  let
    Holding(buffer,next) =
      ( (length(buffer) < N) &
            in?x ->
              sender.send!x.next ->
                Holding(buffer^<x.next>,(next+1)%msn) )
      []

...
```

```
...
        ( (length(buffer) > 0) &
              sender.receive?n ->
                if (n == number(head(buffer))) then
                  Holding(tail(buffer),next)
                else
                  Resend(buffer) ; Holding(buffer,next) )
        []
        ( (length(buffer) == 0) &
              sender.receive?n ->
                Holding(buffer,next) )

      Resend(<>) = SKIP
      Resend(<x>^s) = sender.send!x -> Resend(s)
    within
      Holding(<>,0)
```

```
Receiver(M) =
  let
    Holding(buffer,lastout) =
      ...
      ( receiver.receive?p ->
          if member(number(p),window) then
            Holding(insert(p,buffer),lastout)
          else
            receiver.send!lastout ->
              Holding(buffer,lastout) )
      []
      ( member(next,numbers(buffer)) &
          out!extract(next,buffer) ->
            receiver.send!next ->
              Holding(delete(next,buffer),next)
      ...
```

Coordination

In describing a protocol for coordination or consensus, we may use processes to represent:

- a collection of peer processes;
- some shared resource or communication medium;
- an account of the intended outcome or behaviour.

Mutual Exclusion

A classical problem:

- `P0` and `P1` are two concurrently-executing processes;
- `P0` is capable of performing activity `A0`;
- `P1` is capable of performing activity `A1`;
- `P0` should not perform `A0` while `P1` is performing `A1`, and vice versa—these two activities should be mutually exclusive.

Critical regions

The mutual exclusion problem was originally formulated in terms of concurrently-executing programs—components of an operating system—requiring access to shared resources—such as printers.

A part of a program in which access—an exclusive activity—is performed is called a ‘critical region’.

Programs

```
{P0}
  BEGIN
    .
    .
    <critical region 0>
    .
    .
  END ;
```

```
{P1}
  BEGIN
    .
    .
    <critical region 1>
    .
    .
  END ;
```

Intention

A mutual exclusion algorithm should ensure that no more than one process may be inside a critical region at any one time.

It should do this without introducing the possibility of deadlock; at least one process should be able to proceed.

If a process halts outside its critical region, this should not prevent other processes from proceeding.

Example: doesn't work

```
VAR flag0, flag1 : Boolean
flag0 := false; flag1 := false;
```

```
{P0}
  BEGIN
    flag0 := true;
    WHILE flag1 DO nothing;
    <critical region 0>
    flag0 := false;
  END
```

```
{P1}
  BEGIN
    flag1 := true;
    WHILE flag0 DO nothing;
    <critical region 2>
    flag1 := false;
  END
```

Example

```
PROG = {0..1}
FLAG = {0..1}

channel writeflag, readflag : PROG . FLAG . Bool

channel enter, leave : PROG
```

Example

```
aFlag(this) = {| writeflag.p.this, readflag.p.this | p <- PROG

Flag(this) =
  let
    Status(current) =
      writeflag.this.this?new -> Status(new)
      []
      readflag?prog!this!current -> Status(current)
  within
    Status(False)
```

Example

```
aProg(this) = {| enter.this, leave.this, write.this, read.this

Prog(this,other) =
  let
    Start =
      write.this.this.True -> Wait

    Wait =
      read.this.other.False -> Go

    Go =
      enter.this ->
        leave.this ->
          write.this.this.False -> Start
  within
```

Start

Example

```
FlagEvents = {| readflag, writeflag |}

System =
  ( Flag(0) ||| Flag(1) )
  [| FlagEvents |]
  ( Prog(0,1) ||| Prog(1,0) )

Mutex = enter?i -> leave!i -> Mutex

assert Mutex [T= System \ FlagEvents

assert System :[deadlock free]
```

Dekker’s Algorithm

As an example of a working mutual exclusion protocol, we will consider Dekker’s algorithm.

We will start with a version of the algorithm found in a textbook, then look at an attempted correction, then look at the real thing.

Theodorus Jozef Dekker, born 1 March 1927

Example: found in a book – doesn't work

```
VAR flag0, flag1 : Boolean ; flag0 := false; flag := false;
VAR turn : Id ; turn := 0;
```

```
{P0}
  flag0 := true;
  IF turn = 1 THEN
    WHILE flag1 DO nothing;
  ELSE
    BEGIN
      flag0 := false;
      WHILE turn = 1 DO nothing;
      flag0 := true;
      WHILE flag1 DO nothing;
    END;
  <critical region 0>;
  turn := 1;
  flag0 := false
```

```
{P1}
  flag1 := true;
  IF turn = 0 THEN
    WHILE flag0 DO nothing;
  ELSE
    BEGIN
      flag1 := false;
      WHILE turn = 0 DO nothing;
      flag1 := true;
      WHILE flag0 DO nothing;
    END;
  <critical region 1>;
  turn := 0;
  flag1 := false
```

dekkerB dekkerC

```
Turn(first) =
  let
    Status(current) =
      writeturn?prog?new -> Status(new)
      []
      readturn?prog!current -> Status(current)
  within
    Status(first)
```

```
ProgA(this,other) =
  let
    Start = writeflag.this.this.true -> ReadTurn

    ReadTurn =
      readturn.this.other -> WaitUntilFree
      []
      readturn.this.this -> WaitUntilTurn

    WaitUntilFree = readflag.this.other.false -> Go

    WaitUntilTurn =
      writeflag.this.this.false -> readturn.this.this ->
        writeflag.this.this.true -> WaitUntilFree

    Go =
      enter.this -> leave.this -> writeturn.this.other ->
        writeflag.this.this.false -> Start
  within
    Start
```

progB progC

```
FlagAndTurnEvents = {| writeflag, readflag, writeturn, readturn |}

DekkerA =
  ( Flag(0) ||| Flag(1) ||| Turn(0) )
  [| FlagAndTurnEvents |]
  ( ProgA(0,1) ||| ProgA(1,0) )

assert Mutex [T= DekkerA \ FlagAndTurnEvents

assert DekkerA :[deadlock free[F]]
```

The second check fails.

```
  assert DekkerA \ FlagAndTurnEvents [FD= System \FlagEvents

  assert System \ FlagEvents [FD= DekkerA \FlagAndTurnEvents
```

Both checks succeed: it is exactly the same algorithm!

Example: attempted correction – almost there

```
VAR flag0, flag1 : Boolean ; flag0 := false; flag1 := false;
VAR turn : Id ; turn := 0;
```

```
{P0}
  flag0 := true;
  IF turn = 0 THEN
    WHILE flag1 DO nothing;
  ELSE
    BEGIN
      flag0 := false;
      WHILE turn = 1 DO nothing;
      flag0 := true;
      WHILE flag1 DO nothing;
    END;
  <critical region 0>;
  turn := 1;
  flag0 := false
```

```
{P1}
  flag1 := true;
  IF turn = 1 THEN
    WHILE flag0 DO nothing;
  ELSE
    BEGIN
      flag1 := false;
      WHILE turn = 0 DO nothing;
      flag1 := true;
      WHILE flag0 DO nothing;
    END;
  <critical region 1>;
  turn := 0;
  flag1 := false
```

dekkerA dekkerC

```
ProgB(this,other) =
  let
    Start = writeflag.this.this.true -> ReadTurn

    ReadTurn =
      readturn.this.this -> WaitUntilFree
      []
      readturn.this.other -> WaitUntilTurn

    WaitUntilFree = readflag.this.other.false -> Go

    WaitUntilTurn =
      writeflag.this.this.false -> readturn.this.this ->
        writeflag.this.this.true -> WaitUntilFree

    Go =
      enter.this -> leave.this -> writeturn.this.other ->
        writeflag.this.this.false -> Start
  within
    Start
```

progA progC

```
DekkerB =
  ( Flag(0) ||| Flag(1) ||| Turn(0) )
  [| FlagAndTurnEvents |]
  ( ProgB(0,1) ||| ProgB(1,0) )

assert Mutex [T= DekkerB \ FlagAndTurnEvents
assert DekkerB :[deadlock free[F]]
```

Both checks succeed.

Example: the real thing – from wikipedia

```
VAR flag0, flag1 : Boolean ; flag0 := false; flag1 := false;
VAR turn : Id ; turn := 0;
```

```
{P0}
  flag0 := true;
  WHILE flag1 DO
    BEGIN
      IF turn = 1 THEN
        BEGIN
          flag0 := false;
          WHILE turn = 1 DO nothing;
          flag0 := true;
        END;
    END;
  <critical region 0>;
  turn := 1;
  flag0 := false
```

```
{P1}
  flag1 := true;
  WHILE flag0 DO
    BEGIN
      IF turn = 0 THEN
        BEGIN
          flag1 := false;
          WHILE turn = 0 DO nothing;
          flag1 := true;
        END;
    END;
  <critical region 1>;
  turn := 0;
  flag1 := false
```

dekkerA dekkerB

```
ProgC(this,other) =
  let
    Start = writeflag.this.this.true -> WaitUntilFree

    WaitUntilFree =
      readflag.this.other.false -> Go
      []
      readturn.this.other -> WaitUntilTurn



    WaitUntilTurn =
      writeflag.this.this.false -> readturn.this.this ->
        writeflag.this.this.true -> WaitUntilFree

    Go =
      enter.this -> leave.this -> writeturn.this.other ->
        writeflag.this.this.false -> Start
  within
    Start
```

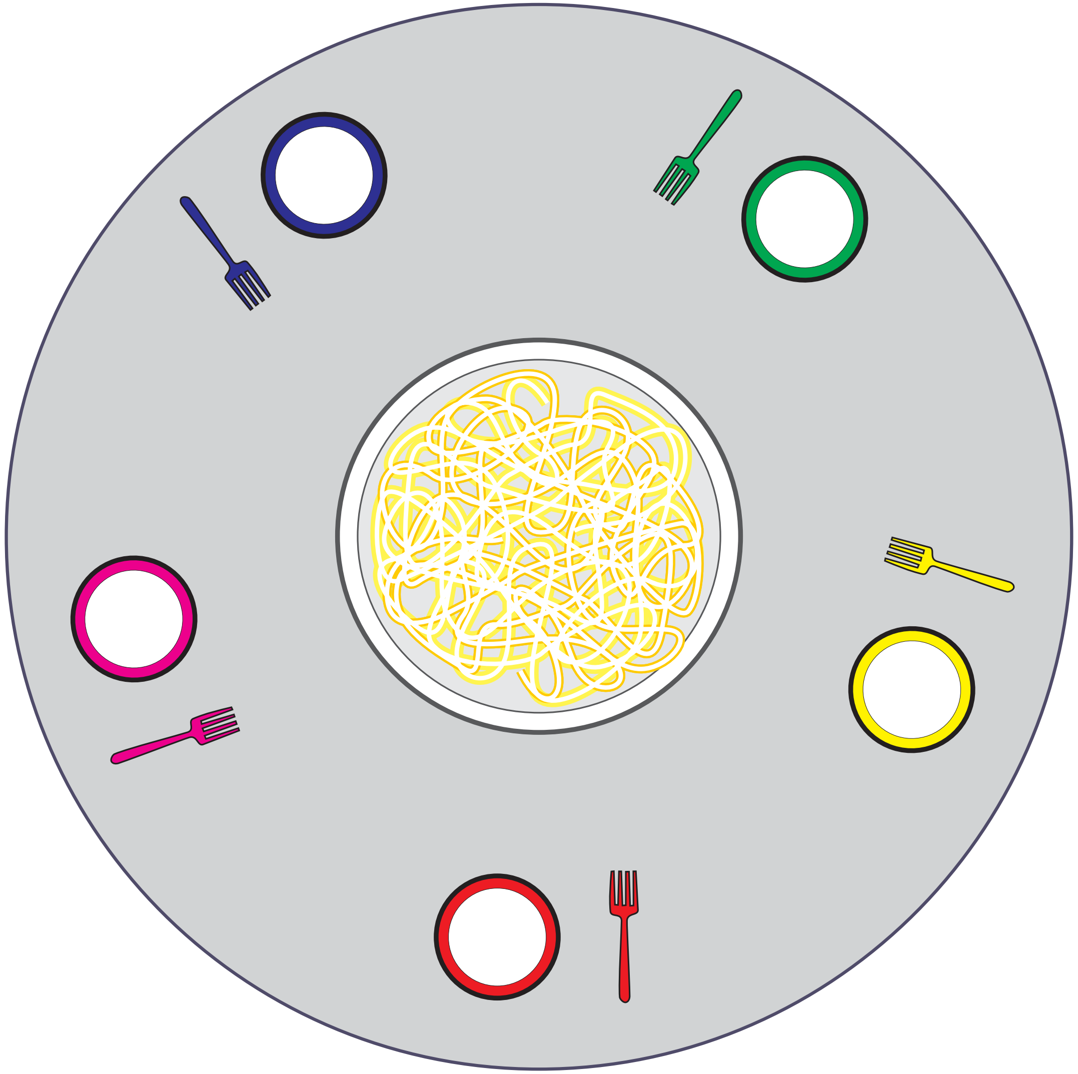
progA progB

```
assert DekkerC \ FlagAndTurnEvents [T= DekkerB \ FlagAndTurnEvents
assert DekkerB \ FlagAndTurnEvents [T= DekkerC \ FlagAndTurnEvents
```

The ‘attempted correction’ version insists that the two programs must alternate. The real thing allows the same program to go again if the other is not waiting.

Dining Philosophers

Five philosophers, five forks, one table, one big bowl of spaghetti



the dining table

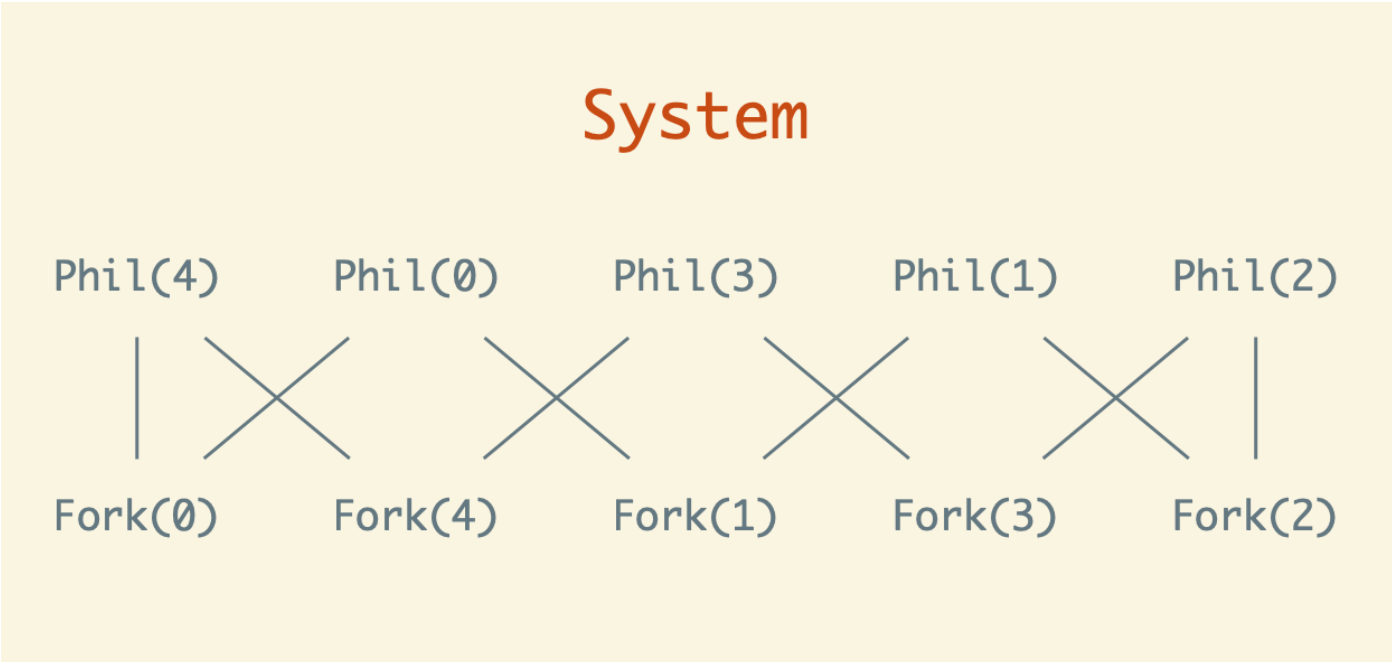
The spaghetti ritual

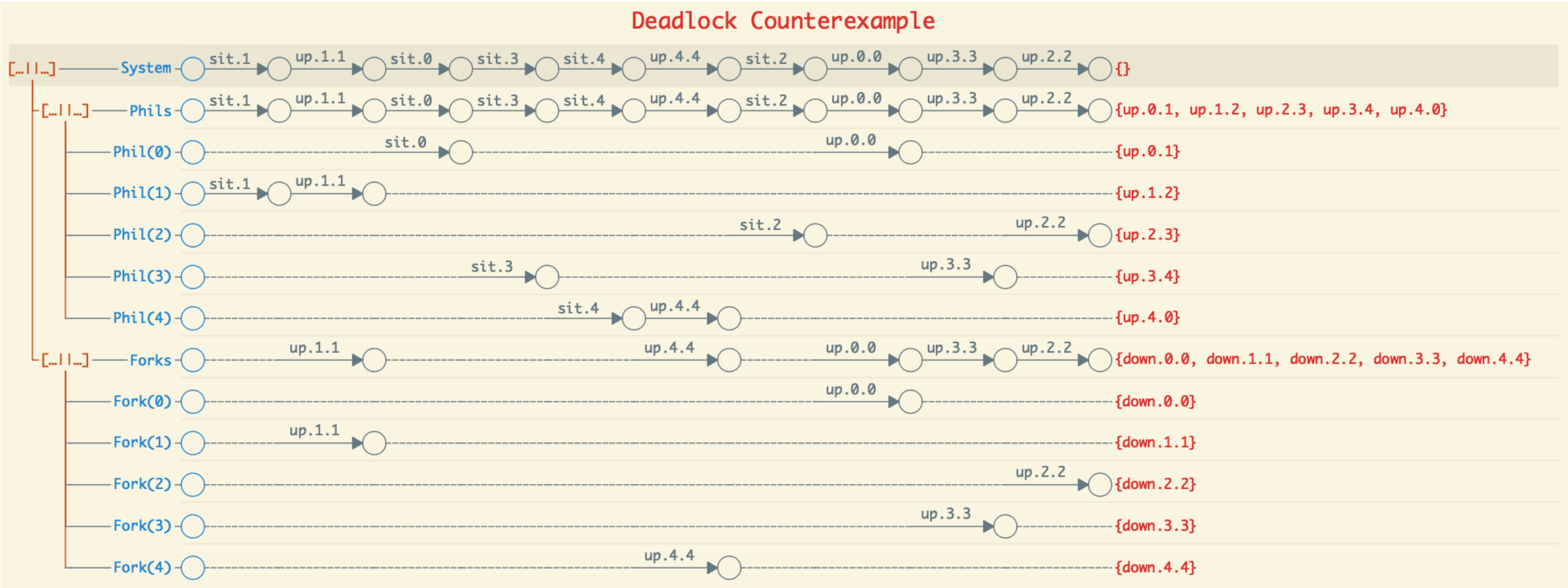
sit down, pick up own fork, pick up neighbour’s fork, (fetch spaghetti), put down neighbour’s fork, (eat), put down own fork, stand

```
FORK = {0..4}
PHIL = {0..4}

channel sit, stand : PHIL
channel up, down : PHIL . FORK

aFork(f) = { up.p.f, down.p.f | p <- PHIL }

Fork(f) = up?p!f -> down!p!f -> Fork(f)
```

```
aPhil(p) = { sit.p, stand.p, up.p.p, up.p.(p+1)%5,
               down.p.(p+1)%5, down.p.p }

Phil(p) =
  sit.p ->
    up.p.p ->
      up.p.(p+1)%5 ->
        down.p.(p+1)%5 ->
          down.p.p ->
            stand.p -> Phil(p)
```

```
aPhils = Union({ aPhil(p) | p <- PHIL})

Phils = || p : PHIL @ [aPhil(p)] Phil(p)


aForks = Union({ aFork(f) | f <- FORK})

Forks = || f : FORK @ [aFork(f)] Fork(f)


System = Phils [ aPhils || aForks ] Forks


assert System :[deadlock free]
```


System
Phil(4)
Phil(0)
Phil(3)
Phil(1)
Phil(2)
Fork(0)
Fork(4)
Fork(1)
Fork(3)
Fork(2)


Deadlock Counterexample
[…||…] System sit.1 up.1.1 sit.0 sit.3 sit.4 up.4.4 sit.2 up.0.0 up.3.3 up.2.2 {}
[…||…] Phils sit.1 up.1.1 sit.0 sit.3 sit.4 up.4.4 sit.2 up.0.0 up.3.3 up.2.2 {up.0.1, up.1.2, up.2.3, up.3.4, up.4.0}
Phil(0) sit.0 up.0.0 {up.0.1}
Phil(1) sit.1 up.1.1 {up.1.2}
Phil(2) sit.2 up.2.2 {up.2.3}
Phil(3) sit.3 up.3.3 {up.3.4}
Phil(4) sit.4 up.4.4 {up.4.0}
[…||…] Forks up.1.1 up.4.4 up.0.0 up.3.3 up.2.2 {down.0.0, down.1.1, down.2.2, down.3.3, down.4.4}
Fork(0) up.0.0 {down.0.0}
Fork(1) up.1.1 {down.1.1}
Fork(2) up.2.2 {down.2.2}
Fork(3) up.3.3 {down.3.3}
Fork(4) up.4.4 {down.4.4}

```
<sit.1, up.1.1, sit.0, sit.3, sit.4, up.4.4, sit.2,
   up.0.0, up.3.3, up.2.2>
```

```
aButler = {| sit, stand |}

Butler =
  let
    Sitting(k) =
      k < 4 & sit?j -> Sitting(k+1)
      []
      k > 0 & stand?j -> Sitting(k-1)
  within
    Sitting(0)
```

```
aSystem = union(aPhils,aForks)

ButlerSystem =
  Butler [ aButler || aSystem ] System

assert ButlerSystem :[deadlock free]
```

Other approaches

- negotiation: additional channels are provided for communication; in case of conflict, a second level of protocol comes into play
- left-handed: one of the philosophers obeys a different rule, always picking up their neighbour's fork first
- expensive: a new fork is purchased.
- extremely expensive: five new forks are purchased
- back-off: in case of conflict, each philosopher will withdraw from the table for a random period of time

Consensus

A consensus protocol allows two or more components—nodes, agents, processes, programs, or systems—to reach agreement upon a particular action or value.

In some cases, it is a simple matter of coordination.

In others, we need to deal with a situation in which one or more of the components may fail or misbehave.

Connection

A simple connection establishment protocol may be used by one client to obtain the agreement of another before data communication begins.

- `send.i.m`: message `m` is transmitted by client `i`
- `receive.i.m`: message `m` is received by client `i`
- `proceed.i`: client `i` proceeds to the data phase of communication
- `abandon.i`: client `i` abandons the attempt at establishing a data connection

The set of messages that may be communicated via the media is given by a free type definition,

```
datatype Message = request | accept | reject
```

Each client is initially ready to transmit or receive a `request` message. Should a `request` be received, the client will stop listening for a `request`, and decide internally whether to reply with an `accept` or a `reject` message.

If it sends an `accept`, it will `proceed` to the data phase; if it sends a `reject`, it will `abandon` the data connection.

A client that transmits a `request` message will then wait for a reply. If it receives an `accept`, it will `proceed` to the data phase; if it receives a `reject`, it will `abandon` the data connection.

```
PROG = {1,2}

datatype MESSAGE = request | accept | reject

channel send, receive : PROG . MESSAGE

channel proceed, abandon : PROG

MediaEvents = {| send, receive |}
```

```
Client(i) =
  let
      Start = send.i.request -> Wait
              []
              receive.i.request -> Reply

       Wait = receive.i.accept -> Proceed
              []
              receive.i.reject -> Abandon

      Reply = send.i.accept -> Proceed
              []
              send.i.reject -> Abandon

    Proceed = proceed.i -> Proceed

    Abandon = abandon.i -> Abandon
  within
    Ready
```

```
SyncModel =
  Client(1)
  [send.1 <-> receive.2, receive.1 <-> send.2]
  Client(2)

Consensus =
  let
    Proceed =
      ||| i : PROG @ proceed.i -> STOP

    Abandon =
      ||| i : PROG @ abandon.i -> STOP
  within
    Proceed |~| Abandon

assert Consensus [FD= SyncModel \ MediaEvents
```

If `send.1` is the same event as `receive.2`, and `send.2` is the same event as `receive.1`, then consensus is guaranteed.

```
Medium(i,j) =  send.i?m -> receive.j!m -> Medium(i,j)

ASyncModel =
  (Client(1) ||| Client(2))
  [| MediaEvents |]
  (Medium(1,2) ||| Medium(2,1))

assert Consensus [FD= ASyncModel \ Internal
```

If not, then consensus is not guaranteed: a deadlock may occur.

A single, shared send-and-receive transaction would be more abstract and easier to work with, but the resulting model would not allow us to address the possibility of a ‘simultaneous open’.

Commit protocols

We can use commit protocols to build transactions out of point-to-point communications—whether these are synchronous or asynchronous.

A commit protocol should guarantee that every party to the transaction will commit, or that every party will cancel.

Two-phase commit

The protocol begins with the coordinator sending a `request` to each of the clients; they may then reply with either `accept` or `reject`.

If every client has accepted, the coordinator sends `confirm` messages to each of them.

If one or more clients rejects the request, then the coordinator sends `cancel` messages instead.

```
Coordinator = InviteAll ; StartListening

InviteAll = ||| c : CID @ invite.c -> SKIP

StartListening = Listening({})
```

```
Listing(A) =
  if A == CID then Confirm
            else ( (accept?c -> Listening(union(A,{c})))
                   []
                   (reject?c -> WillCancel(union(A,{c}))) )

WillCancel(A) =
  if A == CID then Cancel
            else ( (accept?c -> WillCancel(union(A,{c})))
                   []
                   (reject?c -> WillCancel(union(A,{c}))) )
```

```
Confirm = ||| c : CID @ confirm!c -> STOP

Cancel = ||| c : CID @ cancel!c -> STOP
```

```
Client(c) =
  invite.c ->
    ( (accept.c ->
        ( (confirm.c -> Commit(c)
          []
          (cancel.c -> Abort(c))))
      |~|
      (reject.c ->
        ( (confirm.c -> Abort(c))
          []
          (cancel.c -> Abort(c)))))

Commit(c) = commit.c -> STOP

Abort(c) = abort.c -> STOP

Clients = ||| c : CID @ Client(c)
```

```
Messages = {| invite, confirm, cancel, accept, reject |}

System = Coordinator [| Messages |] Clients

Consensus = AllCommit |~| AllAbort

AllCommit = ||| c : CID @ Commit(c)

 AllAbort = ||| c : CID @ Abort(c)

assert Consensus [FD= System \ Messages
```

Three-phase commit

As before, the protocol begins with the coordinator sending a request or invitation to each of the clients.

Each client then replies with an acceptance or a rejection.

The coordinator then informs all clients of the result: that is, whether the intention is to commit or cancel.

Each client then replies with an acknowledgement.

The coordinator then tells each client that they can proceed: they already know whether this means to commit or cancel.

The extra phase means that the clients can recover the situation if the coordinator fails during the process.

see the assignment!

Summary

- Communication
- Coordination
- Consensus

Index

2 Contents
3 Protocols
4 Communication
5 Verification
6 Buffers
7 Example
8 Example
9 Example
16 Lossy Media
17 Retransmit
19 Sliding windows

23 Coordination
24 Mutual Exclusion
25 Critical regions
26 Programs
27 Intention
28 Example: doesn't work
29 Example
30 Example
31 Example
33 Example
34 Dekker's Algorithm
35 Example: found in a book – doesn't work
39 Example: attempted correction – almost there

42 Example: the real thing – from wikipedia

45 Dining Philosophers

47 The spaghetti ritual

56 Other approaches

57 Consensus

58 Connection

64 Commit protocols

65 Two-phase commit

71 Three-phase commit

72 Summary

73 Index