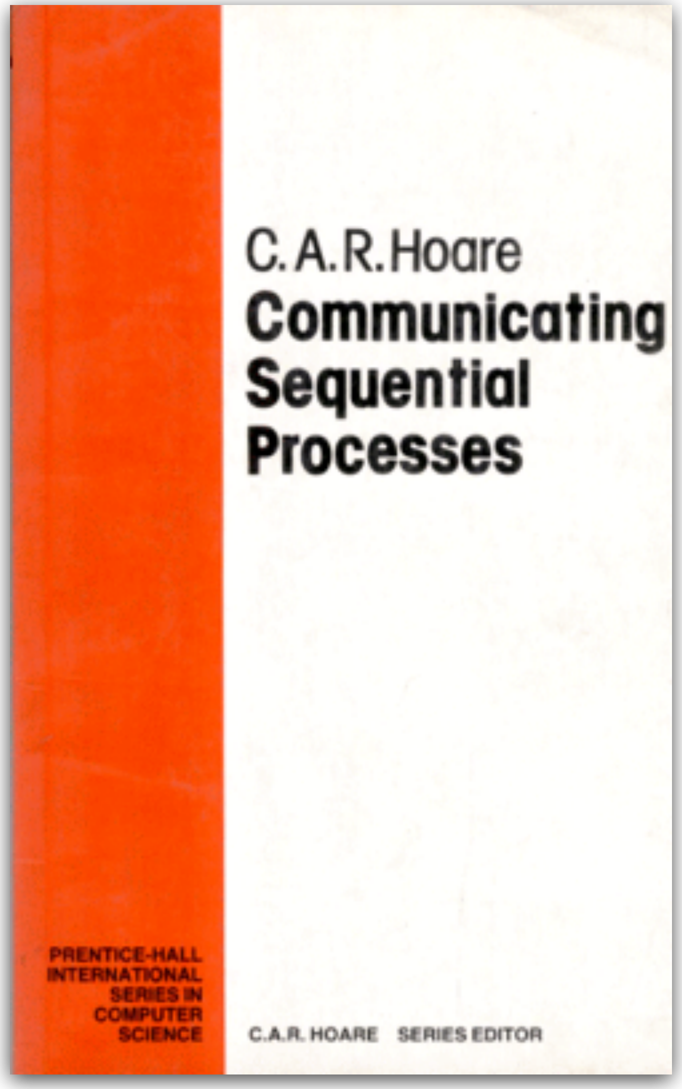
A Channel Compendium

April 24, 2014

John Graham-Cumming

Often mentioned; rarely understood



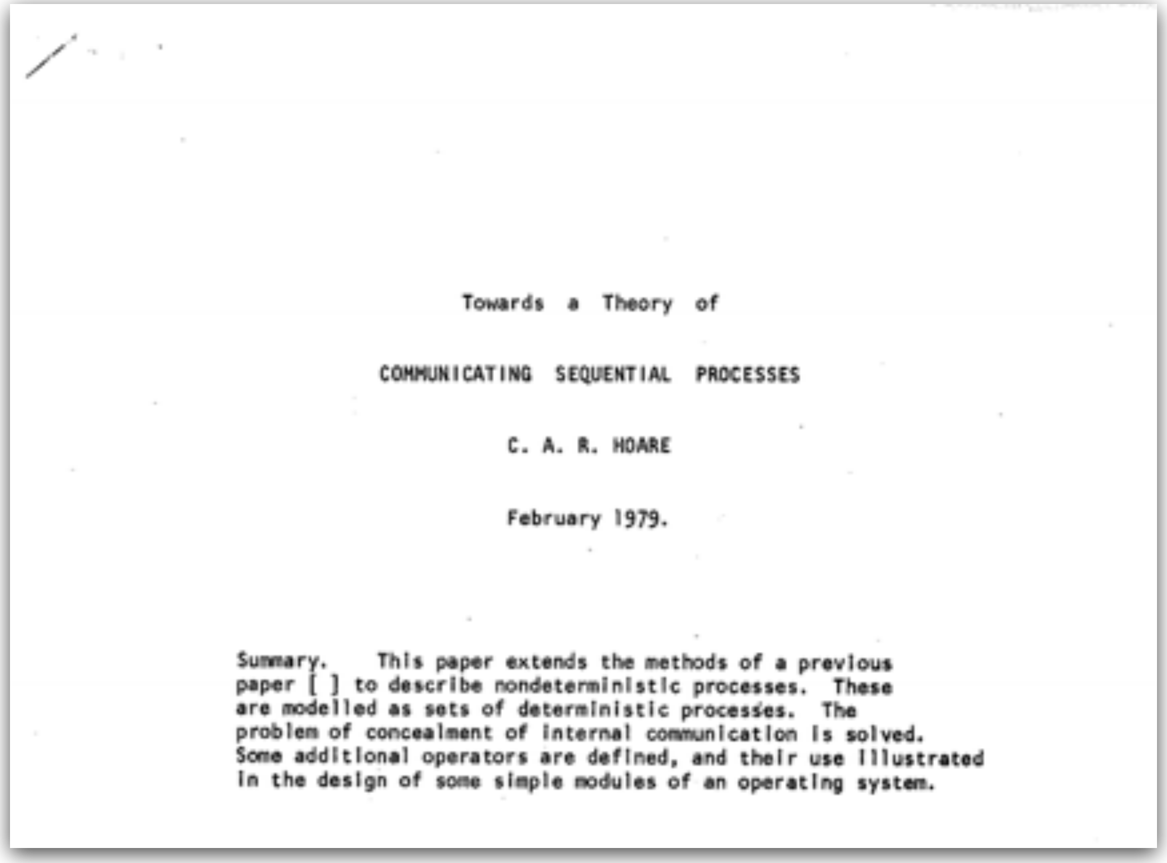
C.A.R.Hoare

Communicating Sequential Processes

PRENTICE-HALL INTERNATIONAL SERIES IN COMPUTER SCIENCE

C.A.R. HOARE SERIES EDITOR

Where channels came from



Towards a Theory of

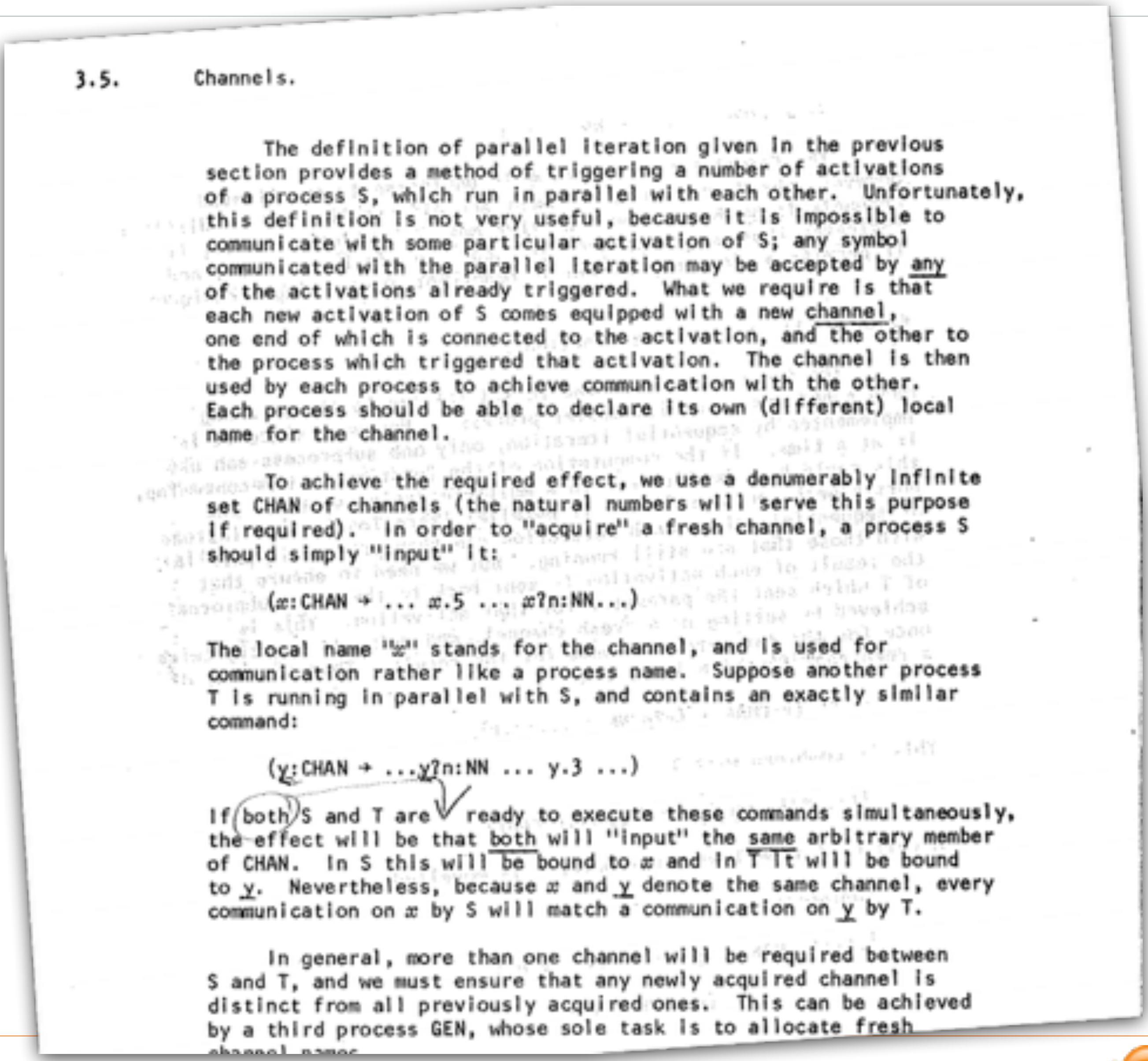
COMMUNICATING SEQUENTIAL PROCESSES

C. A. R. HOARE

February 1979.

Summary. This paper extends the methods of a previous paper [ ] to describe nondeterministic processes. These are modelled as sets of deterministic processes. The problem of concealment of internal communication is solved. Some additional operators are defined, and their use illustrated in the design of some simple modules of an operating system.

Channels



3.5. Channels.

The definition of parallel iteration given in the previous section provides a method of triggering a number of activations of a process S, which run in parallel with each other. Unfortunately, this definition is not very useful, because it is impossible to communicate with some particular activation of S; any symbol communicated with the parallel iteration may be accepted by <u>any</u> of the activations already triggered. What we require is that each new activation of S comes equipped with a new <u>channel</u>, one end of which is connected to the activation, and the other to the process which triggered that activation. The channel is then used by each process to achieve communication with the other. Each process should be able to declare its own (different) local name for the channel.

To achieve the required effect, we use a denumerably infinite set CHAN of channels (the natural numbers will serve this purpose if required). In order to "acquire" a fresh channel, a process S should simply "input" it:

```
(x:CHAN → ... x.5 ... x?n:NN...)
```

The local name "x" stands for the channel, and is used for communication rather like a process name. Suppose another process T is running in parallel with S, and contains an exactly similar command:

```
(y:CHAN → ...y?n:NN ... y.3 ...)
```

If both S and T are ready to execute these commands simultaneously, the effect will be that <u>both</u> will "input" the <u>same</u> arbitrary member of CHAN. In S this will be bound to x and in T It will be bound to <u>y</u>. Nevertheless, because x and <u>y</u> denote the same channel, every communication on x by S will match a communication on <u>y</u> by T.

In general, more than one channel will be required between S and T, and we must ensure that any newly acquired channel is distinct from all previously acquired ones. This can be achieved by a third process GEN, whose sole task is to allocate fresh channel names.

CSP

- Communication == synchronization

$$VendingMachine = coin \rightarrow choc \rightarrow STOP$$

$$Person = (coin \rightarrow STOP) \Box (card \rightarrow STOP)$$

$$VendingMachine \,|[\{coin\}]|\, Person \equiv (coin \rightarrow choc \rightarrow STOP) \Box (card \rightarrow STOP)$$

- When communication and synchronization go together it’s easy to reason about processes

Channels

- Quick syntax review

  `c := make(chan bool)`– Makes an unbuffered channel of `bools`

  `c <- x` – Sends a value on the channel

  `<- c` – Receive a value on the channel

  `x = <- c` – Receive a value and stores it in `x`

  `x, ok = <- c` – Receive a value; `ok` will be false if channel is closed and empty.

Unbuffered channels are best (mostly)

- They provide both communication and synchronization

```
func from(connection chan int) {
    connection <- rand.Intn(100)
}

func to(connection chan int) {
    i := <- connection
    fmt.Printf("Someone sent me %d\n", i)
}

func main() {
    cpus := runtime.NumCPU()
    runtime.GOMAXPROCS(cpus)

    connection := make(chan int)
    go from(connection)
    go to(connection)
…
```

SIGNALLING

Wait for an event

- Sometimes just closing a channel is enough

```
c := make(chan bool)

go func() {
        // ... do some stuff
        close(c)
}()

// ... do some other stuff
<- c
```

- Could replace `close(c)` with `c <- true`

Coordinate multiple goroutines

- Close a channel!

```
func worker(start chan bool) {
    <- start
    // ... do stuff
}

func main() {
    start := make(chan bool)

    for i := 0; i < 100; i++ {
        go worker(start)
    }

    close(start)

    // ... all workers running now
}
```

Select

- Select statement enables sending/receiving on multiple channels at once

```
select {
case x := <- somechan:
    // ... do stuff with x

case y, ok := <- someOtherchan:
    // ... do stuff with y
    // check ok to see if someOtherChan
    // is closed

case outputChan <- z:
    // ... ok z was sent

default:
    // ... no one wants to communicate
}
```

Common idiom: `for/select`

```
for {
    select {
    case x := <- somechan:
        // ... do stuff with x

    case y, ok := <- someOtherchan:
        // ... do stuff with y
        // check ok to see if someOtherChan
        // is closed

    case outputChan <- z:
        // ... ok z was sent

    default:
        // ... no one wants to communicate
    }
}
```

Terminate workers

- Close a channel to terminate multiple goroutines

```
func worker(die chan bool) {
    for {
        select {
            // ... do stuff cases
        case <- die:
            return
        }
    }
}

func main() {
    die := make(chan bool)
    for i := 0; i < 100; i++ {
        go worker(die)
    }
    close(die)
…
```

Verify termination

- Terminate a goroutine and verify termination

```
func worker(die chan bool) {
    for {
        select {
            // ... do stuff cases
        case <- die:
            // ... do termination tasks
            die <- true
            return
        }
    }
}
func main() {
    die := make(chan bool)
    go worker(die)
    die <- true
    <- die
}
```

Closed channels never block

```
func main() {
    c := make(chan bool)
    close(c)
    x := <- c
    fmt.Printf("%#v\n", x)
}
```

x has the zero value for the channel's type

```
func main() {
    c := make(chan bool)
    close(c)
    x, ok := <- c
    fmt.Printf("%#v %#v\n", x, ok)
}
```

ok is false

```
func main() {
    c := make(chan bool)
    close(c)
    c <- true
}
```

Closing buffered channels

```
func main() {
    c := make(chan int, 3)
    c <- 15
    c <- 34
    c <- 65
    close(c)

   fmt.Printf(“%d\n”, <-c)
   fmt.Printf(“%d\n”, <-c)
   fmt.Printf(“%d\n”, <-c)

   fmt.Printf(“%d\n”, <-c)
}
```

Drains the buffered data

Starts returning the zero value

range

- Can be used to consume all values from a channel

```
func generator(strings chan string) {
    strings <- "Five hour's New York jet lag"
    strings <- "and Cayce Pollard wakes in Camden Town"
    strings <- "to the dire and ever-decreasing circles"
    strings <- "of disrupted circadian rhythm."
    close(strings)
}

func main() {
    strings := make(chan string)
    go generator(strings)

    for s := range strings {
        fmt.Printf("%s ", s)
    }
    fmt.Printf("\n");
}
```

HIDE STATE

Example: unique ID service

- Just receive from `id` to get a unique ID
- Safe to share `id` channel across routines

```
id := make(chan string)

go func() {
    var counter int64 = 0
    for {
        id <- fmt.Sprintf("%x", counter)
        counter += 1
    }
}()

x := <- id // x will be 1
x = <- id  // x will be 2
```

Example: memory recycler

```
func recycler(give, get chan []byte) {
    q := new(list.List)

    for {
        if q.Len() == 0 {
            q.PushFront(make([]byte, 100))
        }

        e := q.Front()

        select {
        case s := <-give:
            q.PushFront(s[:0])

        case get <- e.Value.([]byte):
            q.Remove(e)
        }
    }
}
```

DEFAULT

`select` for non-blocking receive

A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case b = <-idle:

default:
    makes += 1
    b = make([]byte, size)
}
```

Try to get from the idle queue

Idle queue empty? Make a new buffer

`select` for non-blocking send

A buffered channel makes a simple queue

```
idle:= make(chan []byte, 5)

select {
case idle <- b:

default:
}
```

Try to return buffer to the idle queue

Idle queue full? GC will have to deal with the buffer

NIL CHANNELS

`nil` channels block

```
func main() {
    var c chan bool
    <- c
}
```

```
func main() {
    var c chan bool
    c <- true
}
```

`nil` channels useful in `select`

```
for {
    select {
    case x, ok := <-c1:
       if !ok {
            c1 = nil
       }

    case x, ok := <-c2:
       if !ok {
            c2 = nil
       }
    }
    if c1 == nil && c2 == nil {
        return
    }
}
```

Works for sending channels also

```
c := make(chan int)
d := make(chan bool)

go func(src chan int) {
       for {
              select {
              case src <- rand.Intn(100):

              case <-d:
                     src = nil
              }
       }
}(c)

fmt.Printf("%d\n", <-c)
fmt.Printf("%d\n", <-c)
d <- true
fmt.Printf("%d\n", <-c)
```

TIMERS

Timeout

```
func worker(start chan bool) {
     for {
         timeout := time.After(30 * time.Second)
         select {
                 // ... do some stuff

           case <- timeout:
                 return
           }
     }
}
```

```
func worker(start chan bool) {
     timeout := time.After(30 * time.Second)
     for {
         select {
                 // ... do some stuff

           case <- timeout:
                 return
           }
     }
}
```

Heartbeat

```
func worker(start chan bool) {
    heartbeat := time.Tick(30 * time.Second)
    for {
       select {
            // ... do some stuff

        case <- heartbeat:
            // ... do heartbeat stuff
        }
    }
}
```

EXAMPLES

Example: network multiplexor

- Multiple goroutines can send on the same channel

```
func worker(messages chan string) {
    for {
        var msg string // ... generate a message
        messages <- msg
    }
}
func main() {
    messages := make(chan string)
    conn, _ := net.Dial("tcp", "example.com")

    for i := 0; i < 100; i++ {
        go worker(messages)
    }
    for {
        msg := <- messages
        conn.Write([]byte(msg))
    }
}
```

Example: first of N

- Dispatch requests and get back the first one to complete

```
type response struct {
    resp *http.Response
    url string
}

func get(url string, r chan response ) {
    if resp, err := http.Get(url); err == nil {
        r <- response{resp, url}
    }
}

func main() {
    first := make(chan response)
    for _, url := range []string{"http://code.jquery.com/jquery-1.9.1.min.js",
        "http://cdnjs.cloudflare.com/ajax/libs/jquery/1.9.1/jquery.min.js",
        "http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js",
        "http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.9.1.min.js"} {
        go get(url, first)
    }

    r := <- first
    // ... do something
}
```

Passing a ‘response’ channel

```
type work struct {
    url string
    resp chan *http.Response
}

func getter(w chan work) {
    for {
        do := <- w
        resp, _ := http.Get(do.url)
        do.resp <- resp
    }
}

func main() {
    w := make(chan work)

    go getter(w)

    resp := make(chan *http.Response)
    w <- work{"http://cdnjs.cloudflare.com/jquery/1.9.1/jquery.min.js",
        resp}

    r := <- resp
}
```

Example: an HTTP load balancer

- Limited number of HTTP clients can make requests for URLs
- Unlimited number of goroutines need to request URLs and get responses

- Solution: an HTTP request load balancer

A URL getter

```
type job struct {
    url string
    resp chan *http.Response
}

type worker struct {
    jobs chan *job
    count int
}

func (w *worker) getter(done chan *worker) {
    for {
        j := <- w.jobs
        resp, _ := http.Get(j.url)
        j.resp <- resp
        done <- w
    }
}
```

A way to get URLs

```
func get(jobs chan *job, url string, answer chan string) {
    resp := make(chan *http.Response)
    jobs <- &job{url, resp}
    r := <- resp
    answer <- r.Request.URL.String()
}

func main() {
    jobs := balancer(10, 10)
    answer := make(chan string)
    for {
        var url string
        if _, err := fmt.Scanln(&url); err != nil {
            break
        }
        go get(jobs, url, answer)
    }
    for u := range answer {
        fmt.Printf("%s\n", u)
    }
}
```

A load balancer

```
func balancer(count int, depth int) chan *job {
    jobs := make(chan *job)
    done := make(chan *worker)
    workers := make([]*worker, count)

    for i := 0; i < count; i++ {
        workers[i] = &worker{make(chan *job,
            depth), 0}
        go workers[i].getter(done)
    }

    go func() {
        for {
            var free *worker
            min := depth
            for _, w := range workers {
                if w.count < min {
                    free = w
                    min = w.count
                }
            }

            var jobsource chan *job
            if free != nil {
                jobsource = jobs
            }
            select {
            case j := <- jobsource:
                free.jobs <- j
                free.count++

            case w := <- done:
                w.count—
            }
        }
    }()

    return jobs
}
```

THANKS

The Go Way: “small sequential pieces joined by channels”