A Model of Distributed Systems

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Distributed Systems

Early on, we defined a distributed system as:

> … *multiple computer programs, possibly spread out over different networked components, communicating by passing messages*

What are:

- computer programs?
- communication?
- messages?

What is our model of distributed systems?

Computer Programs

What is a computer program? is a hard question.

We will take an abstract view:

- A sequence of instructions
- Performing some task

For our purposes, these multiple programs could be:

- Built from the same source code
- Built from different source code
- Threads in a single process
- Separate processes possibly on different computers

Message Passing

There are many avenues for message passing:

- Shared memory
- Files
- Sockets
- Pipes
- Go channels

This is distinct from general shared state, however.

Many programming models can be implemented through message passing.

Concurrency

In this model, many programs run concurrently.

This means that multiple programs may appear to make progress simultaneously.

From the perspective of a program P:
Between time t and $t + \epsilon$, a program Q may take some action.

Whether P and Q actually run simultaneously is irrelevant! [4]

Synchronous Systems

In a synchronous system, all actions take predictable time:

- A message sent from P to Q always arrives within some bounded time.
- The relative rate of progress in P and Q is known.

Examples of synchronous systems are:

- Symmetric multiprocessor computers
- Some circuit-switched networks

Some tasks are substantially easier in synchronous systems.

We usually will not examine synchronous systems.

Asynchronous Systems

In an asynchronous system, actions take unpredictable time:

- Messages may be arbitrarily delayed
- The difference in rate of progress in different processes is unbounded

All Internet protocols are asynchronous.

Asynchronous systems have special challenges.

We will focus on asynchronous systems.

Implications of Asynchrony

Asynchronous systems present challenges.

Suppose that:

- P sends a message to Q and expects a response.
- No message arrives for longer than expected.

What happened?

- Did Q fail?
- Is Q much slower than P expects, and still working?
- Was the request message delayed in the network?
- Was the request message lost?
- Was the response delayed or lost?

Loss and Delay

Loss and delay are indistinguishable in an async system.

You cannot tell whether a message is:

- late, or
- never going to arrive.

TCP and Loss

Recall that TCP used heuristics to detect loss!

1. A relatively long time without positive acknowledgment
2. Acknowledgment of some data but not all

There is no reliable loss indicator.

The missing segment might just be stuck in a queue somewhere!

Loss *as* Delay

In the end, sometimes loss looks like delay.

Consider what happens if TCP loses a segment:

- The next data cannot be delivered at the receiver
- Eventually the sender retransmits
- The data is delivered at the receiver later than expected

In particular:
Loss at a lower layer may look like delay at a higher layer.

Loss and Failure

Loss and failure may also be indistinguishable.

This is a consequence of the system relying on message passing.

Consider:

- Process *P* sends a message to *Q* and expects a reply
- *P* never receives a reply

Did *Q* fail (crash, shut down, *etc.*)?

Was *P*'s message lost, or *Q*'s reply lost?

P can't tell.

Correctness and Safety

The introduction of concurrency has implications on correctness.

Operations that are safe without concurrency may become unsafe.

Example:
Suppose we have a variable `x = 0` visible to both P and Q.
P : `x = x + 1`
Q : `x = x - 1`

If these execute concurrently, what is `x`?

We don’t have enough information.

Race Conditions

This is a race, or race condition:

- Two or more events are dependent upon each other
- Some of the events may happen in more than one order, or even simultaneously
- There exists some ordering of the events that is incorrect

For example:

- Some state will be updated multiple times
- Output will be produced based on the state

If some order of updates results in invalid output, this is a race.

Example Race

P : `x = x + 1`
Q : `x = x - 1`

There are at least three possible outcomes here:

- `x = -1`
- `x = 0`
- `x = 1`

Why?

Atomicity

These statements are not atomic: they can be interrupted.

`x = x + 1` is at least three operations:

- Read the value of `x`
- Add one to that value
- Write the new value to `x`

P reads `x`	
	Q reads `x`
P computes `x + 1`	*Q* computes `x - 1`
P stores `x = x + 1`	
	Q stores `x = x - 1`

Happens Before

The happens before relationship ensures a particular outcome.

If `x = x + 1` happens before `x = x - 1`, then `x = 0`.

By judicious use of happens before, we can prevent races.

Many languages define happens before relationships.

The Go Memory Model [5] defines this for Go.

Mutexes

Mutexes can be expressed as happens before relationships.

From the Go memory model:

> *For any* `sync.Mutex` *or* `sync.RWMutex` *variable* `l` *and* `n < m`*, call* `n` *of* `l.Unlock()` *happens before call* `m` *of* `l.Lock()` *returns.*

These guarantees must be made explicit in a language!

You cannot assume happens before relationships.

Messages

A message send happens before its corresponding receive.

This is trivially true for a network transmission.

This is guaranteed by Go channels.

In shared memory, use mutexes or other synchronization.

Communicating Sequential Processes

Tony Hoare proposed communicating sequential processes in 1978 [1].

CSP is a programming model built on message passing.

Hoare showed that it can:

- Model other constructions (such as subroutines)
- Enable parallel computation
- Naturally express concurrent problems

CSP in Distributed Systems

CSP maps naturally to distributed systems:

- Distributed systems communicate by message passing
- Message exchanges create happens before relationships

Many distributed systems languages and libraries emulate CSP.

Go channels implement CSP input and output operations.

Socket communications can also provide CSP input and output.

Fixing x

With CSP, we can ask a single process to manipulate `x`:

```
func handleX() {
    for cmd := range c
       {
        switch cmd {
        case INCREMENT:
            x = x + 1
        case DECREMENT:
            x = x - 1
        }
    }
}
```

```
func P() {
    c <- INCREMENT
}

func Q() {
    c <- DECREMENT
}
```

Summary

- Distributed systems communicate by message passing
- We will work with asynchronous systems
- Delay is indistinguishable from loss
- Concurrent execution can lead to races
- Happens before is the cure for races
- CSP is a programming model for message passing

Next Time . . .

- Failures and failure detection

References I

Required Readings

[3] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Chapter 1: 1.1–1.3, 1.5–1.8. Cambridge University Press, 2008. ISBN: 978-0-521-18984-2.

Optional Readings

[1] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 666–677. URL: `https://search.lib.buffalo.edu/permalink/01SUNY_BUF/12pkqkt/cdi_crossref_primary_10_1145_359576_359585`.

References II

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. URL: `http://www.usingcsp.com/`.

[4] Rob Pike. *Concurrency is not Parallelism*. Jan. 2012. URL: `https://go.dev/blog/waza-talk`.

[5] *The Go Memory Model*. May 2014. URL: `https://go.dev/ref/mem`.