Concurrency and the real world

Richard Carlsson

Hardware and software

Modern hardware

- The frequency race is long since over
- Power consumption and heat is a big issue
- Many cores running on reasonable frequencies
- RAM is cheap, but coherence doesn't scale
- Disks are really slow, even SSD
- You typically have lots of different machines
  - Load balancers, SSL termination, ...
  - Front end servers, database servers, authentication, ...
  - Logging, backup, monitoring, ...

Modern software

- To do more work, you need to split up your code!
- Software is adapting very slowly to multicore
  - Desktop software only does it in special cases
    - Photo, video, sound processing, etc. easy to parallellize
  - Games: rendering, AI, physics, etc.; still much to do
    - Imagine running the AI for each NPC as a separate thread
  - Server software is easiest to parallellize
    - Requests from clients can be handled independently
    - One thread per client connection – possibly more
    - But the code that handles a request is usually sequential

Legacy software

- You rarely get to start completely from scratch
  - When you do, you don't think that your code might live for 10-20 years and will need to scale 1000 times
- Code becomes legacy as soon as the person who wrote it has moved on to other things
- Parallelizing code can be hard - but it's 10 times harder if you don't fully understand what it's doing
  - Side effects make code hard to understand/change
  - Code that relies on a shared memory space must be completely rewritten to run on separate machines

Understanding concurrency

What is time?

- You need to relearn how to think about time
  - It takes experience – you have to build a new intuition
- Clocks are mostly useless; just as in relativistic physics, there is no global ”right now”
  - Only messages and replies can be relied on
    - Causal relationships: if B, then A must have happened
    - Logical time: discrete points, partial order
  - Even the smallest step in your program can take anything from a nanosecond to several hours

What is concurrency?

- If two things can *potentially happen at the same time*, they are said to be concurrent
  - Maybe they usually happen in a certain order, but if a delay happens somewhere, the order could change
- Delays can happen at any time, and for any length of time (CPU scheduling, memory stalls)
- The behaviour under real load will not be like in development - if the program can fail, it will!
  - ”Fixing” a timing problem by adding a pause is broken from start, and will come back to bite you later

What is parallelism?

- Utilizing multiple resources in parallel
  - CPUs, GPUs, RAMs, data buses, interfaces, disks
- Concurrency is needed for parallelism
  - A completely non-concurrent program has nothing that can be done in parallel at any time
- Parallelism is not needed for concurrency
  - A concurrent program can use a single resource by interleaving the separate tasks in small portions

Resources and bottlenecks

- Whenever concurrent tasks want to use the same limited resource, you get a bottleneck
  - A narrow passage that everything has to go through
- No task can make progress until it has passed the bottleneck
  - Processes fighting over CPU or RAM resources
  - Processes fighting over disk access or network bandwidth
  - Threads killing each other's cache behaviour

Designing for concurrency

- Keep it simple!
  - Knowing that it works is always more important
- Think about your program as a set of services
  - Within each service, let each activity be a process
  - No process should know more than it needs to know
  - Always keep in mind that you may want to split out the different parts to run on separate machines
  - Avoid traditional ”object-oriented” design, which can lead to a spaghetti of shared data dependencies

Programming techniques

Fail fast and noisily

- The program should fail immediately on errors
  - The longer it tries to keep going, the harder it will be to figure out what really went wrong
- Check inputs at the borders of your program
  - Internally, assume that the inputs are correct
- Let it crash!
  - Don't try to detect errors and ”autocorrect” them – crash the program and let someone else restart it
- Write to proper logs, don't print to stdout!

Separate all the things

- Traditional optimizations try to do as much as possible at the same time, in a single thread
  - For example, loop over all items in an array and do a whole bunch of different things to each item
  - This makes it impossible to utilize multicore, since you cannot easily split the code into separate processes
- ”Optimizing” by doing several unrelated things in the same piece of code, just because you have the information available, is a really bad idea

Controlling your parallelism

- Handling a million requests in parallel is great. Making a million mistakes in parallel is bad!
  - Bugs, resource limits (file descriptors, memory)
  - If one worker fails, the others will probably fail too
    - For one thing, this can flood your logs
  - Circuit breaker pattern
- Make features configurable: runtime on/off switch
- Log Pids or session IDs so you can follow what a session has done, across services

Splitting shared resources

- When parallelizing, you will often find a few central things that all your processes need to access
  - Typical example: global counters (session IDs, etc.)
  - Performance bottleneck – everything gets serialized
  - Single point of failure – if it goes offline, nothing works
- Must eliminate the sharing. Think outside the box.
  - Global server handing out number series to each front end server on demand, enough for hours or days
  - Does the application require guaranteed uniqueness or just very low probability of collisions? Use a hash!

Explicit sequencing

- You sometimes find that parts of your system need to be explicitly serialized to avoid race conditions
- Typically, when more than one part can make active decisions. E.g., a hotel booking site:
  - 2 customers see room is available and press ”book”
  - As long as one of them gets an error message and has to try again, it's all right
- Often handled by database transactions, but you may need to do it all by yourself
  - E.g., send all such requests through a single server

Know what is important

- Service Level Agreement: ”we promise that...”
- Which of your services *must* always be available to keep customers happy, and which are ”extras”?
  - Don't let non-critical stuff bring your system down
- You need to know which errors are critical, and which are just ”bad” but can be fixed
  - Sold a book that was out of stock? Mail customer about delay, order more books from publisher.
  - Two customers booking same room is bad, but two planes on the same runway must simply never happen.

Persistent storage

Most systems need a database

- Database = global shared memory = bad!
  - All of the code assumes it can access all of the data
  - Often leads to bad code with lots of dependencies
- Databases usually imply transactions
  - Transactions imply bottlenecks (locks or collisions)
  - You have to weigh consistency against bottlenecks
- A file system is no exception: it's a kind of database (hierarchical, good at ”file-ish” things, bad at other things, usually no transactions)

Databases and distribution

- To guarantee availability, you need more than one machine, and more than one copy of your data
- The famous ”CAP theorem”
  - Consistency, Availability, Partition tolerance
    - You can't have all 3 at the same time - pick 2 (CP/AP)
    - You *will* get partitioning; in a real system, networks fail
    - You mostly *need* availability, or you will be out of business
- Transactions need to (mostly) go away
  - Eventual consistency – allow temporary inconsistency
  - No simple rules for writing transactionless systems

Erlang – built for concurrency

Erlang design philosophy

- Isolation of processes, no shared memory, message passing, error handling via links
  - A process can't corrupt the state of another process
- Mostly functional programming, few side effects (messages), easy to read and reason about
- Divide your program into small components, no shared state
- Service oriented software design

Erlang and multicore

- The Erlang philosophy is a great fit for multicore
  - No need for locks or synchronized sections
  - No shared memory – a process can run anywhere
- Erlang gives you built-in support for multicore, but can also run the same concurrent code on a single-core machine
- Erlang lets you run processes on separate machines over the network just as easily as on a single machine, without rewriting any of the code

Why not use ... instead?

- Virding's law: ”Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang.”
- Erlang lets you get close enough to the best possible performance in a very short time
  - The code is clean and easy to modify further
- If your program is not really concurrent, or needs shared memory, another language might be better.

Final words

Concurrency is hard – or not

- Most people aren't used to thinking about concurrent programming – it takes some practice
- On the other hand, as a human you deal with concurrency every day
  - Drinking coffee and answering a mail while talking to someone; not bumping into people in the corridor; watching TV and eating cheetos while texting a friend...
- Erlang is great for experimenting and getting a feeling for concurrency – play around!

We live in exciting times

- Few textbook examples of how to solve typical problems without resorting to transactions
- Huge demand for systems that scale well and remain available despite partial failures
- Most situations need a combination of engineering and domain specific workarounds
- High performance parallel (scientific) computing, distributed web services, games, AI and robotics, etc., have very different requirements. Pick a language that suits the problem.

We're always hiring...

klarna.com/jobs