abstractions at scale

our experiences at twitter

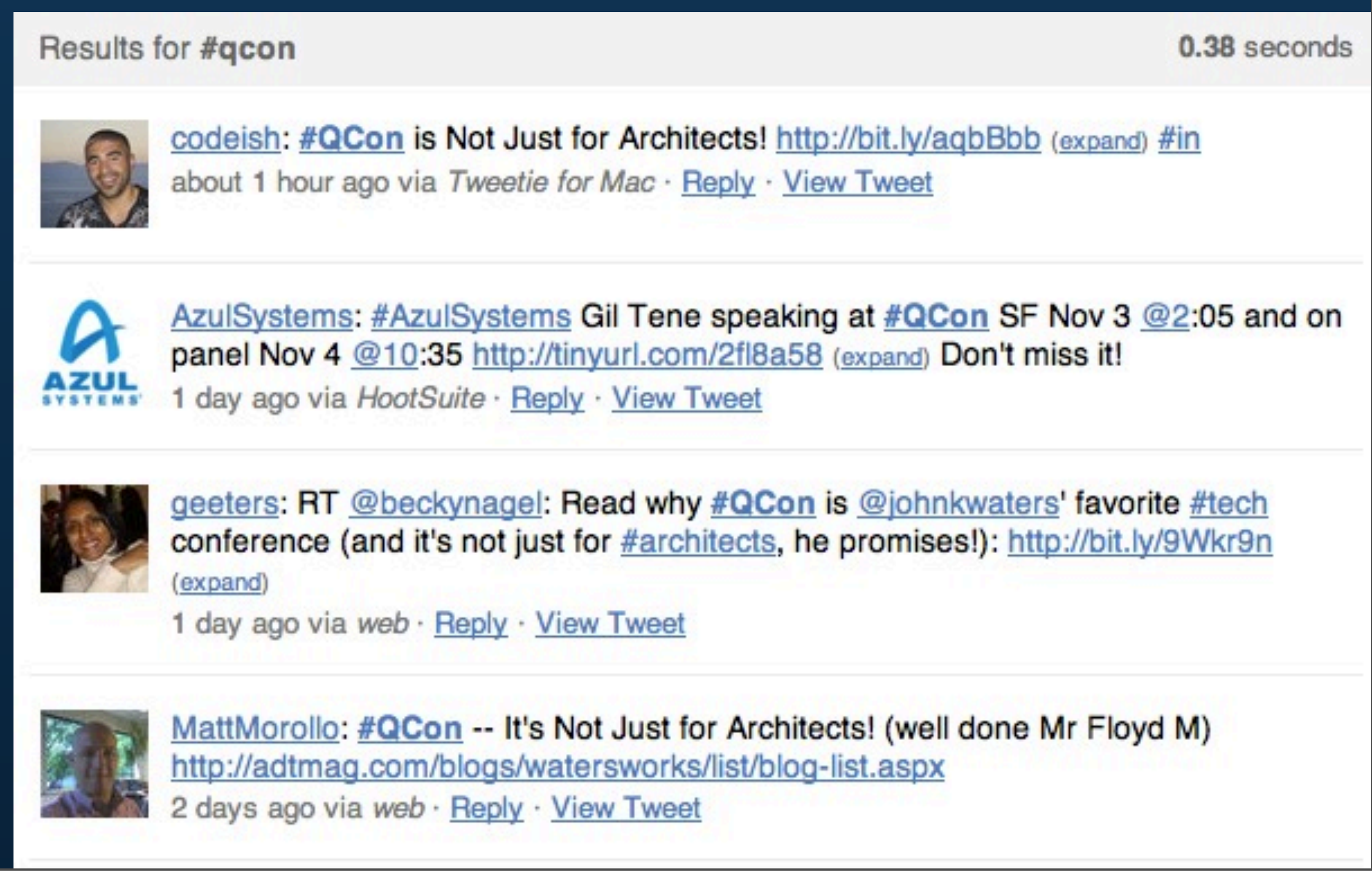
marius a. eriksen
@marius

QConSF, November 2010

twitter™

twitter

- real-time information network
  - 70M tweets/day (800/s)
  - 150M users
  - 70k API calls/s



agenda

- scale & scalability
- the role of abstraction
- good abstractions, bad abstractions
- abstractions & scale
- examples
- “just right” APIs
- conclusions

scale & scalability

"*Scalability* is a desirable property of a system, a network, or a process, which indicates its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged"

(Wikipedia)

scale & scalability (cont'd)

- only "horizontal" scaling allows unbounded growth
  - not entirely true: eg. due to network effects
  - not a panacea
- "vertical" scaling is often desirable & required
  - contain costs
  - curtail network effects

scale & scalability (cont'd)

- the *target architecture* is the datacenter
  - network is a critical component
  - deeper storage hierarchy
  - higher performance variance
  - complex failure modes
  - but our programming models don't account for these resource & failure models explicitly

abstraction

"freedom from representational qualities"

- the chief purpose of abstraction is to manage complexity & provide composability
- in software, abstraction is manifested through common *interfaces*
  - explicit semantics
  - implicit "contracts"

abstraction (cont'd)

- as systems become more complex, abstraction becomes increasingly important
  - especially as number of engineers grow
- modern systems are highly complex *and are highly abstracted*

type systems

- [static] type systems can encode *some* of the contracts for us
  - giving us *static* guarantees
  - academia is pushing the envelope here with dependent types
  - they also compose
- the line between type & program becomes blurred

good abstraction? your CPU

- x86-64 is a **spec**
  - you don’t care if it’s provided by AMD or Intel
- excepting a few compiler & OS authors, most of you don’t think about
  - pipelining
  - out of order & speculative execution
  - branch prediction
  - cache coherency
  - etc…

good ...? your memory hierarchy

- you don't interface with it directly
- purist view: addressable memory cells
- reality: has scary-good optimizations for common access patterns. *highly optimized.*
- you don't think (often) about:
  - cache locality
  - TLB effects
  - MMU ops scheduling

bad abstraction? ia64

- *(at least initially)* compilers couldn't live up to it
  - *hardware* promise was delegated to the *compiler*
  - compilers failed to reliably produce sufficiently fast code
  - abstraction was broken
  - good for certain scientific computing domains

a lens

- scaling issues occur when abstractions become leaky
  - RDBMS fails to perform sophisticated queries on highly normalized data
  - your GC thrashes after a certain allocation volume
  - OS thread scheduling becomes unviable after N × 1000 threads are created

¶ threads

- threads offer a familiar and linear model of execution
  - scheduling overhead becomes important after a certain amount of parallelism
  - stack allocation can become troublesome
  - fails to be explicit about latency, backpressure
- alternative: asynchronous programming
  - makes queuing, latency explicit
  - allows SEDA-style control
- a compromise? LWT

¶ sequence abstractions

- produces concise, beautiful, composable code

```
trait Places extends Seq[Place]
places.chunks(5000).map(_.toList).parForeach { chunk =>
  …
}
```

- access patterns aren’t propagated down the stack
- missed optimizations

¶ RDBMS

- are [by definition] generic
- encourage normalized data storage
  - *very powerful* data model
  - little need to know access patterns a priori
- provide general (magical) querying mechanics
  - bag of tricks: query planning, table statistics, covering indices

¶ RDBMS

- at scale, the most viable strategy is: **W**hat **Y**ou **S**erve **I**s **W**hat **Y**ou **S**tore (WYSIWYS)
  - or at least very close
- this brings about a whole host of *new* problems
  - data (in)consistency
  - multiple indices
  - “re-normalization”

¶ RDBMS

- at-scale, querying is *highly predictable*, most of the time:
  - don’t need fancy query planning
  - don’t need statistics
- in fact, we know a-priori how to efficiently query the underlying datastructures
  - wish: don’t give me a query engine, give me primitives!
  - maybe there’s a “just right” API here

¶ in-memory representations

- having tight control over representation is often crucial to resource utilization
  - [space vs. time] memory bandwidth is precious, CPU is plentiful
  - cache locality can often make an *enormous* difference — even to the point of less code is better than more efficient code(!)
- at odds with modern GC'd languages automatic memory management & layout

¶ in-memory representations

- optimize memory layout
- pack data
- compression
  - varint, difference, zigzag, etc.
- L1:main memory latency ≈ 1:200 (!)

- example: geometry of Canada ~ jts normalized, vs. WKB
  - wkb is ≈ 600 KB, JTS representation ≈ 2-3MB

¶ garbage collection

- we *love* garbage collection
- attempts to encode common patterns: generational hypothesis
  - not always quite right
  - the application almost always has some idea about object lifetime & semantics
- proposal: talk to each other!
  - backpressure, thresholding, application-guided GC

¶ virtual memory

“You’re Doing it Wrong”

Poul-Henning Kamp, ACM Queue, June 2010

"… Varnish does not ignore the fact that memory is virtual; it actively exploits it”

¶ virtual memory

- maybe *he* is doing it wrong?
- varnish uses data structures designed to anticipate virtual memory layout & behavior
  - translates application *semantics* (eg. LRU)
- instead, you could have *direct control* over those resources

“just right” abstractions

- high level abstractions are absolutely necessary to deal with today’s complex systems
- but providing *good* abstractions is hard
- what are the “just right” abstractions?
  - exploit common patterns
  - give enough degrees of freedom to the underlying platform
  - usually target a narrow(er) domain
  - retain high level interfaces

¶ mapreduce

```
def map(datum):
  words = {}
  for word in parse_words(datum):
    word[word] += 1
  for (word, count) in words.items():
    output(word, count)

def reduce(key, values):
  output(key, mean(values))
```

- much freedom is given to the scheduler
- exploits data locality (predictably)

¶ shared-nothing web apps

```
def handle(request):
  return Response(
    “hello %s!” % request.get_user())
```

- eg: google’s app engine, django, rails, etc

¶ bigtable

- very simple data model
  - but composable — effectively every other database squeezes (more) sophisticated data models down to 1 dimensional storage(s)
- explicit memory hierarchy (pinning column families to memory)
- provides load balancer/scheduler much freedom
- only magic: compactions. challenge: resource isolation.

¶ LWT

```
lwt ai = Lwt_lib.getaddrinfo
  "localhost" "8080"
  [Unix.AI_FAMILY Unix.PF_INET;
   Unix.AI_SOCKTYPE Unix.SOCK_STREAM] in

lwt (input, output) =
  match ai with
    | [] -> fail Not_found
    | a :: _ -> Lwt_io.open_connection
                  a.Unix.ai_addr in

Lwt_io.write output "GET / HTTP/1.1\r\n\r\n" >>
Lwt_io.read input
```

theme

- provide a programming model that provide a narrow (but flexible) interface to resources
  - mapreduce
  - shared-nothing web apps
- provide a programming model that make *resources* explicit
  - bigtable
  - LWT

meta pattern(s)

- addressing separation of concerns:
  - (asynchronous) execution policy vs. (synchronous) application logic
  - data locality vs. data operations
  - data model vs. data distribution
  - data locality vs. data model

the future?

- database systems
- search systems
- ... or any online query system?

- some academic work already in this area:
  - OPIS (distributed arrows w/ combinators)
  - ypnos (grid compiler)
  - skywriting (scripted dataflow)

conclusions

- we *need* high level abstractions
  - they are simply necessary
  - allows us to develop faster and safer
- many high level abstractions aren't "just right"
  - can become highly inoptimal (often orders of magnitudes can be reclaimed)
- some systems *do* provide good compromises
  - makes resources explicit
- the future is exciting!

that’s it!

- follow me: @marius
- marius@twitter.com