Distributed Systems and the End of the API

Chas Emerick

@cemerick @QuiltProject

PhillyETE
April 23, 2014

Preface

- I have no turnkey solutions (yet!)
  - Objective: to point towards consensus about the problems in our systems today, and emergent strategies on how to address them in the future
- Why should you listen to me?

Claims

- The notion of the networked application API is an unsalvageable anachronism that fails to account for the necessary complexities of distributed systems.
- There exist a set of distributed systems formalisms that do account for these complexities, but which are effectively absent from modern programming practice.

Definitions

Distributed Systems

- (Almost) every system is a distributed system
- "A distributed system is one where a machine I've never heard of can cause my program to fail."
- Any system comprised of multiple processes that must communicate to perform work
- Uniformly unintuitive semantics around causality, consistency, and availability
- If you haven't accepted that you're building one, your luck will inevitably run out

“Application Programming Interface”

- Originally coined in the context of imperative programming languages
  - e.g. Win32, POSIX, all the names in your favorite javadoc
- Long since overloaded to refer to sets of named operations offered by network application integration paths
- Strictly <u>nominal</u> description of classes/modules/methods providing imperative operations; no formalism of operations' semantics

The narrowness of "APIs"

- RPC $\rightarrow$ #{DCOM, CORBA} $\rightarrow$ RMI $\rightarrow$ XML-RPC $\rightarrow$ SOAP $\rightarrow$ REST $\rightarrow$ #{"REST", Thrift}
  - These are all fundamentally equivalent
- Inescapable programming language heritage:
  - request/response
  - fundamentally synchronous
  - point-to-point communication topology
  - nearly always imperative (mutable data models & side-effecting operations)
  - Few constraints on data models or representations

Chas Emerick
@cemerick

People hear "RPC", and giggle, smugly shaking their head while pounding out REST integrations.

Reply Delete Favorite Buffer ••• More

RETWEETS	FAVORITES
17	11

12:33 PM - 7 Feb 2014

— https://twitter.com/cemerick/status/431843135904571392

APIs → Sisyphean programmer convenience

- Primary focus is maintaining isomorphism to method calls:

```
api.create(arg1, arg2);

POST http://site.org/resource/create
[arg1, arg2]

PUT http://site.org/resource
[arg1, arg2]
```

- Ironic that providers of e.g. HTTP APIs often also offer "client libraries" for various languages that restore the classic RPC programming experience
  - Continuing to lull us all into thinking that we're just making regular ol' same-process method calls

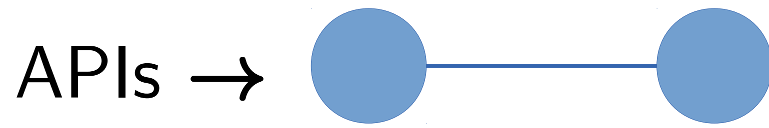
— http://this-plt-life.tumblr.com/post/36425228283/when-somebody-asks-me-how-i-added-a-dsl-to-my-lisp

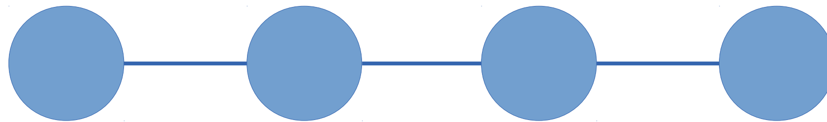
The API is an anachronism

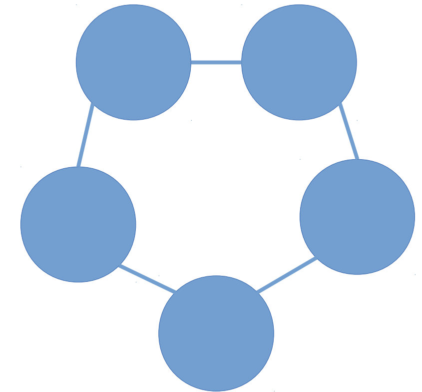
- Intense coupling between client and server
- Forces a two-party client/server architecture, despite realities
- Network failure modes and common computational tasks necessitate asynchrony
- Disavows the fundamental complexities of distributed systems
  - Failure modes
  - Availability considerations
  - Consistency choices
  - Data model characteristics

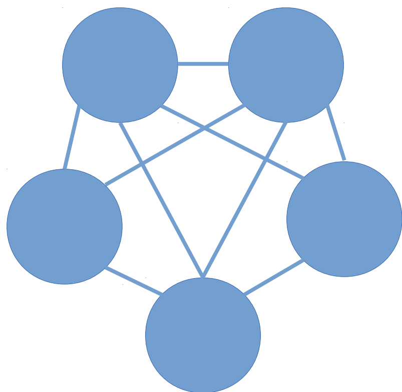
There's more than one system topology

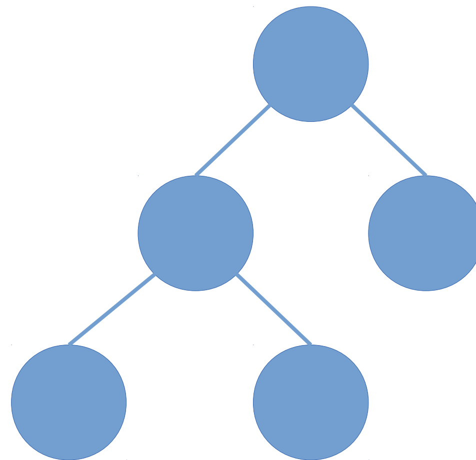
APIs →

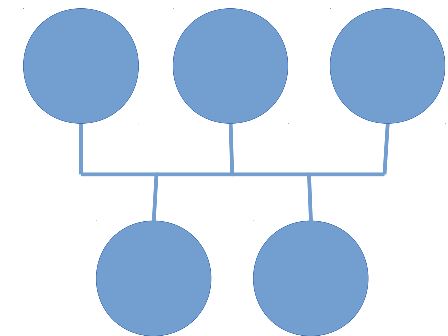










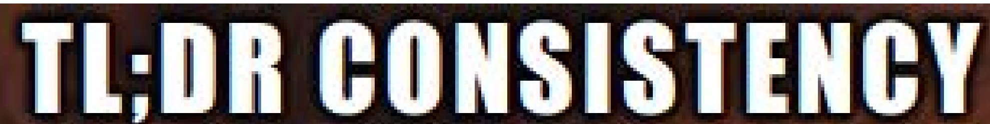


Acknowledge the network or fail

- Failure modes
  - Partitions
    - Complete loss of interconnect
    - Offline operation
    - Variable latency
  - Reordered messages
  - Repeated messages
- Your network's problems are your system's problems
- My network's problems are your system's problems

Consistency decisions affect *everything*

- Degrees of consensus yield degrees of consistency
  - Synchronous acknowledgement of each write by all actors $\rightarrow$ strict linearizability (global total order of all operations)
  - Tracking temporal relationships between dependent data $\rightarrow$ causal consistency (read your own writes)
  - Concurrent writes converge such that different readers will see the results of the last write at some point in the future $\rightarrow$ eventual consistency
- The choices made here will dictate your system's availability characteristics


TL;DR CONSISTENCY
YOU MUST CHOOSE...WISELY

What do we want?

Communication

Computation

We've been here before

- assembly/C : Java/Python/Clojure :: APIs : ???
- Just as our predecessors identified problems with machine code and assembly and constructed abstractions in higher-level languages, we must rise above the metal (sockets, RPC, etc)
- Much of this work is identifying what *not* to do, just as higher-level languages are largely characterized by their constraints (no JMPs, no direct memory access, no in-place mutation of objects)

Appeal to Authority

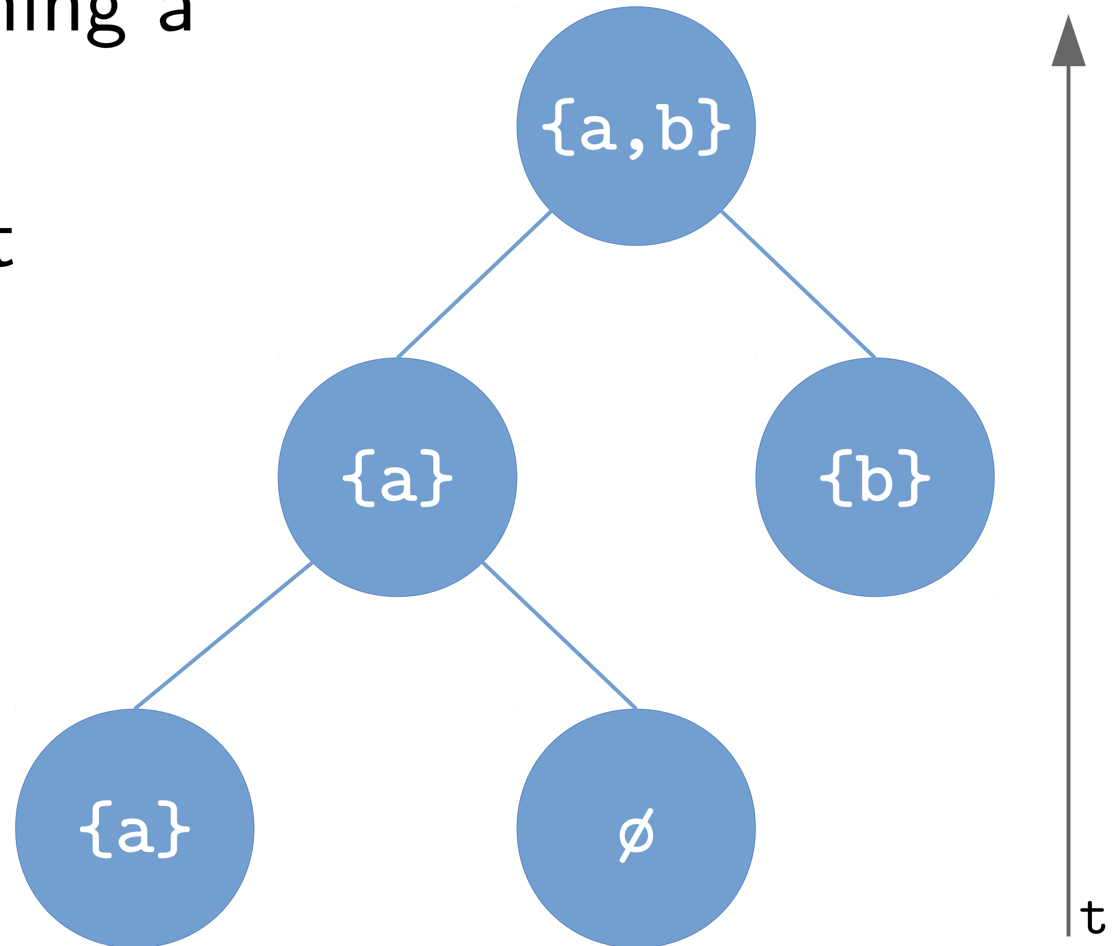
Leslie Lamport

Sound approaches

- Consistency As Logical Monotonicity (CALM theorem)
- Conflict-free Replicated Data Types (CRDTs)
- Constraining the types of operations in order to:
  - Ensure convergence of changes to shared data by uncoordinated, concurrent actors
  - Eliminate network failure modes as a source of error
- ~~Clever implementation details~~
- Math!

bounded-join semilattices

- join: operation defining a least upper bound
- Partially-ordered set
- *Always "increasing", adding information*



The math is really easy

- If your data structure's operations satisfy three basic properties, it's a semilattice.
  - Associativity
    `f(f(a, b), c) = f(a, f(b, c))`
  - Commutativity
    `f(a, b) = f(b, a)`
  - Idempotence
    `f(f(a)) = f(a)`
- Semilattice data structures are immune to messages being lost, reordered, or delivered multiple times

Data models are everything

- Semilattices *expand* the set of data structures you can use in a distributed context
  - Counters, registers, sets, (multi)maps, trees, graphs, vectors
- Immutability meshes perfectly with semilattice semantics, yields treble benefits in a distributed system
  - Histories, rollbacks, consistent snapshots come for free
- The API use cases? Reify operations into data.
  - Instead of calling `api.setName(personId, "Chas")`, merge `{:person-id person-id :name "Chas"}` into a CRDT
  - "Operations" are now *computable*, just like any other data: copy them, route them, reorder them freely, at any level of your system
  - Many of the advantages of queues flow from their forcing exactly the same transformation

Have N programming models

- Event sourcing and other log-structured approaches
- Stream-based computation
- Tuple spaces & other blackboard systems
  - Linda
  - reactive patterns brought to distributed computation
  - Operations triggered in response to data arriving that matches a pattern / satisfies a query
- RPC if necessary, but reify operations into data
- New languages that assume these semantics
  - Bloom

What do we want?

- Communication
- Computation
- Services (and people!) reactively manipulating a shared substrate of replicated data
  - Supersets all use cases for APIs as currently construed
  - No coordination or coupling between actors
  - Allows for arbitrary computational models and application/network topologies

Resources

- Chris Meiklejohn's 'Readings in Distributed Systems': http://bit.ly/cmeik-dist-sys-readings
- Bloom, a Ruby DSL for "disorderly programming": http://www.bloom-lang.net
- CRDTs offered in v2.0 of Riak: http://bit.ly/riak-crdts
- The Quilt Project: http://quilt.org

Thank you, Philly!

@cemerick @QuiltProject