KubeCon | CloudNativeCon

North America 2019

Design Decisions for Communication Systems

Eric Anderson – Google; gRPC

KubeCon | CloudNativeCon

North America 2019

Into the Rabbit Hole

We’re going on an adventure!

Programming Languages

Engineers familiar with multiple programming languages

Engineers *opinionated* about programming languages

Programming Languages

Are you interested in a language that is:

- Imperative
- Strongly-typed
- Dynamically-type-checked
- Object-oriented
- Garbage-collected
- JITed
- Memory-safe
- Multi-threaded

Programming Languages

Are you interested in a language that is:

- Imperative
- Strongly-typed
- Dynamically-type-checked
- Object-oriented
- Garbage-collected
- JITed
- Memory-safe
- Multi-threaded

- With lambdas

Communication Systems

Engineers *opinionated* about communication systems

Communication Systems

Engineers *opinionated* about communication systems…

in similar way as *emacs* vs *vim*

Why the few options?

Communication Systems

What are the choices for how to communicate?

Let’s see…

- REST
- RPC
- Proprietary protocol
- ???

Communication Systems

What are the choices for how to communicate?

Let’s see…

- Request/response vs… ???
- Client-server vs… ???
- Binary vs text

Communication Systems

What are the choices for how to communicate?

Let’s see…

- Request/response vs… ???
- Client-server vs… ???
- Binary vs text

Is that really all there is?

Starting simple

Pipe

Ya know, that Unix thing
You can send and receive

Pipe

Simplex
Reliable
Ordered
Byte-oriented
Streaming

Asynchronous
Flow controlled
Buffered
Anonymous
Serial

Pipe

Simplex (vs duplex)

- Only one direction
- Except it is duplex (both directions) in some OSes

Reliable (vs unreliable)

Ordered (vs unordered)

Byte-oriented (vs message-oriented)

Streaming

- Any number of elements (bytes), with an end (tends to imply reliable and ordered)

Asynchronous (vs synchronous)

- Sender does not wait for reader

Pipe

Flow controlled

- Reader limits send rate

Buffered (vs unbuffered)

- Provides performance. Related to async

Anonymous (vs named)

- There is no way to “find” a pipe; you must be given the pipe fd to use it

Serial (vs parallel)

- Only one sender and receiver at a time for multi-byte
- Is partially parallel for single-byte

Pipe

Frequently two are paired together

Duplex

- Two-direction

Full duplex (vs half duplex)

- Both sides can send at any time

Point-to-point (in practice)

Proven tool, even though slightly low-level and local-only

FIFO (named pipe)

Named (vs anonymous)

- Is a file that can be opened

The pipe is still “one time use.” After it is closed, the file is useless and just be deleted

Shared Resources

Implicit communication via

- Shared memory
- Shared memory+mutex
- File
- File+file locks
- RDMA

Shared Resources

- Common for desktop applications
- Common for intra-app communication
- High performance

- Brittle, but adding restrictions makes manageable
    - Poorly suited for crossing trust domains
    - Poorly suited to outgrow a single specific job

Common patterns, but will be application-specific protocol

Shared Resources

Bit too complex and varied to get into

When scaling over many machines, can still share resources via a network protocol

- Many interaction patterns still hold

Sockets

- Duplex stream of bytes or messages
- Point-to-point
- Client-server
  - The server binds to a port or name that the client knows to connect to
- Connection-oriented

Unix Domain Socket (bytes or messages)
TCP (bytes)

Messages may have a maximum size

Sockets

UDP (messages)

- Except it isn’t ordered
- Except it doesn’t guarantee delivery
- Except it doesn’t have flow control
- Except it isn’t connection-oriented
- Except it can multicast to multiple receivers
- Yeah… let’s stop talking about UDP

Unix Domain Socket

Allows transferring system objects (e.g., FDs)

- Commonly used to limit permissions

Unix Domain Socket

Allows transferring system objects (e.g., FDs)

- Commonly used to limit permissions

How are system objects' lifetime managed?

Commonly reference-counted by the kernel

- FDs don't hold open other FDs, so "flat" reference counting system; no graph, no cycles

Higher-level protocols

RPC

Remote Procedure Call
SunRPC; json-rpc; SOAP; gRPC

- Request/response messages
- Point-to-point
- Client-server
- Connectionless
- IDL: Interface Definition Language
- Generated Code
- Synchronous

RPC

Remote Procedure Call
SunRPC; json-rpc; SOAP; **gRPC**

- Request/response **and streaming** messages
- Point-to-point
- Client-server
- Connectionless
- IDL: Interface Definition Language
- Generated Code
- Synchronous **(req/resp) and async (streaming)**

RPC

Remote Procedure Call
SunRPC; json-rpc; SOAP; **gRPC**

Streaming allows pipelining

- Something between serial and parallel

RPC?

```
service Creator {
  rpc Create(Empty) returns (CreateResponse);
}
message CreateResponse {
  Calculator calc = 1;
}
service Calculator {
  rpc Add(AddRequest) returns (AddResponse);
}
```

RMI

Remote Method Invocation. “Object-oriented RPC”

Object

- State with associated methods
- Passed by reference

Message

- Just data. Primitives and structs
- Passed by value

Implications of References

Need a way to “bootstrap”

- Directory service where objects “bind” to names
- Returned objects need be casted

Need a way to define methods

- Have “services” that are interfaces
- Runtime type system to query interfaces of objects

Need a way to manage object lifetime

- Need reference counting/GC

D-Bus example

```
bus = dbus.SystemBus()
avahi_proxy = bus.get_object(
        "org.freedesktop.Avahi", "/")
server = dbus.Interface(
        avahi_proxy,
        "org.freedesktop.Avahi.Server")
# Actual communication
browser = server.ServiceBrowserNew(...)
```

Implications of References

Need a way to “bootstrap”

- Directory service where objects “bind” to names
- Returned objects need be casted

Need a way to define methods

- Have “services” that are interfaces
- Runtime type system to query interfaces of objects

Need a way to manage object lifetime

- Need reference counting/GC

Implications of References

Need a way to “bootstrap”

- Directory service where objects “bind” to names
- Returned objects need be casted

Need a way to define methods

- Have “services” that are interfaces
- Runtime type system to query interfaces of objects

Need a way to manage object lifetime

- Need reference counting/GC

RMI

- Request/response *objects* and messages
- Point-to-point
- Not *plain* client-server
- Connectionless
- IDL: Interface Definition Language
- Generated Code
- Synchronous

And sometimes:

- Network transparency

RMI

- Android Binder
- D-Bus

- DCOM
- CORBA
- Java RMI

Local RMI allows transferring system objects (e.g., FDs)

- The reference itself is a “secret”
- Commonly used to limit permissions

KubeCon | CloudNativeCon

North America 2019

Small Detour

Brokered

Uses an intermediary, the “broker”; is a service

- Message/job queue
- Message bus
- Watcher/notification

Not client/server; is its own topology

- But is generally built on a client/server protocol
- D-Bus is built on Unix Domain Sockets
- Google Pub/Sub is built on gRPC

Brokered

Uses an intermediary, the “broker”; is a service

- Message/job queue
- Message bus
- Watcher/notification

Well suited for one-way communication
Well suited for “multicast”
Performance strongly dependent on use case and the broker implementation

</Detour>

HTTP/REST

- Loosely object-oriented
  - Resource URIs are references (“http://host/ref”)
  - Methods (GET, PUT, DELETE) are applied to resources
  - References can be passed in and returned
  - References sometimes used for security; but often not
- Very few methods
  - Additional resources, content-type, and request contents used to define more specific interfaces
- Sometimes uses IDL
- No reference counting/GC
  - Transient objects rare

REST

K8s uses REST
Actually has a GC

- Resources are a bit different from objects

HTTP/REST

- Byte-based streaming available
  - Half-duplex
  - Client-streaming commonly unavailable
- Pipelining abandoned

- Virtual hosting
- L7 routing

- Caching
  - Proxies commonly used. Generally unsupported in client libraries

HTTP/REST vs RPC

REST “OO” is mainly in the application. Exceptions:

- L7 routing
  - Routing is typically limited to the resource structure
  - Could route based on other information in headers
- Caching, only applies to GET
  - Could use RPC request as key

Interesting they aren’t more different

Non-functional

Implementation quality
Ease of use
IDL maintainability
Ecosystem compatibility
Ecosystem size
Debuggability
Performance
Efficiency
…

Q&A

GitHub/Gitter: @ejona86
Email: ejona@google.com (please CC mailing list)
Mailing List: grpc-io@googlegroups.com
Stack Overflow: #grpc #grpc-java

Wednesday 12:25pm
Meet the gRPC Maintainers at the Google Cloud Community Lounge

KubeCon | CloudNativeCon

North America 2019

Request/Response vs… ???

- One-way (fire-and-forget)
- Message queue
- Message bus
- Watcher/notification
- Streaming
- Shared memory
- RDMA

Client-server vs… ???

This strongly influences the system’s topology

- Peering
- Message queue
- Bus
- Pipeline