Learning the Three Types of Microservices

Mike Amundsen
@mamund
API Academy


Mike Amundsen
@mamund

http://apiacademy.co

http://g.mamund.com/msabook

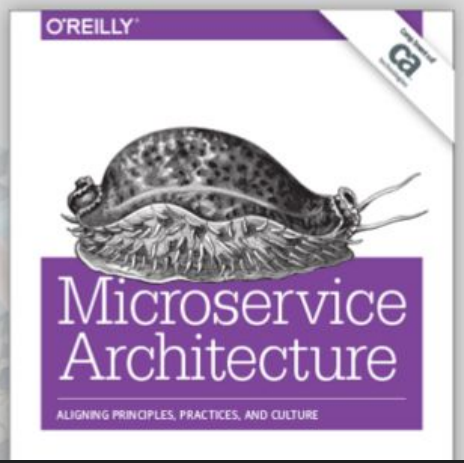
ca technologies

Microservice Architecture: Aligning Principles, Practices, and Culture

Microservices is the next evolution in software architecture designed to help organizations embrace continual change in the digital economy. But how do you design and apply an effective microservice architecture?

This new book from O'Reilly provides comprehensive guidance through seven valuable chapters that give you a deep-dive into:

- The benefits and principles of microservices
- A design-based approach to microservice architecture
- Lessons for applying microservices in practice



Overview

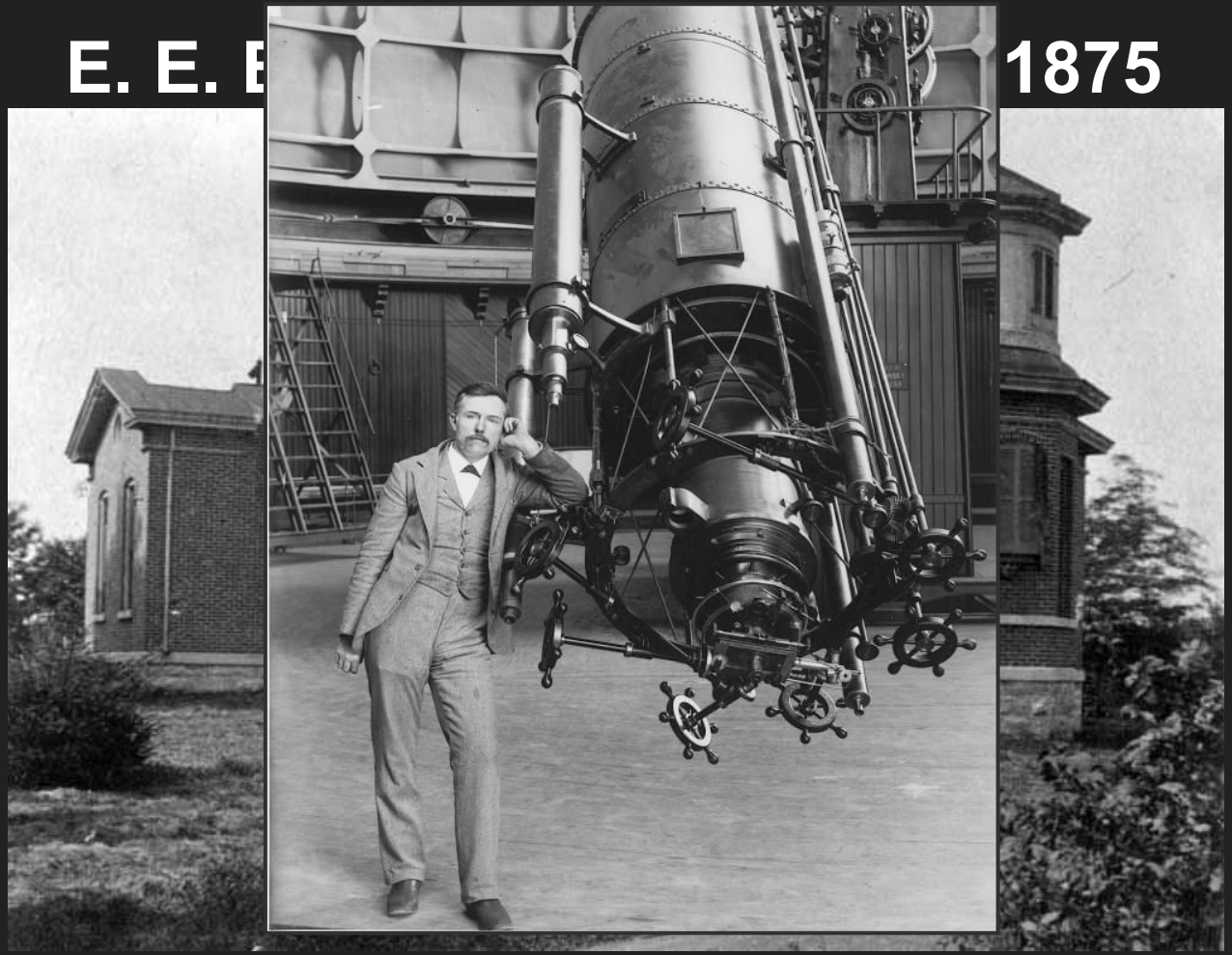
- Programming the Network
- Microservices
- Three Types of MSC
- Nygard's Stability Patterns
- Applying Nygard to MSA
- But Wait, There's More...


SARRATT STUDENT

E. E. Barnard Observatory, 1875

http://www.library.vanderbilt.edu/speccol/exhibits/barnard/vanderbilt.shtml

E. E. B 1875

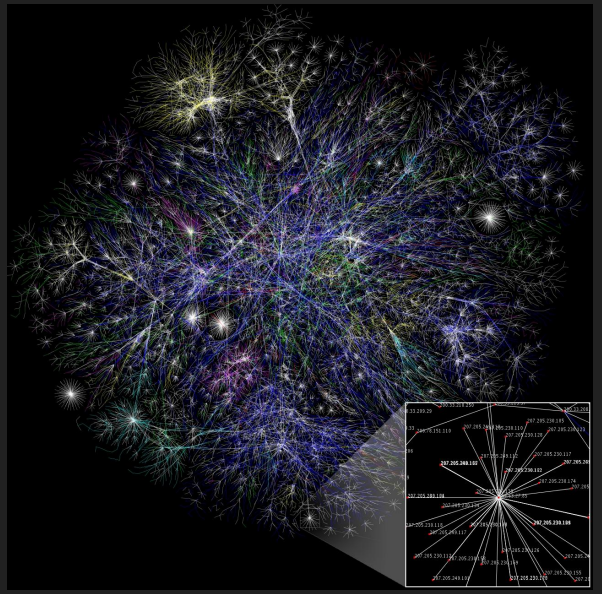


http://www.library.vanderbilt.edu/speccol/exhibits/barnard/vanderbilt.shtml

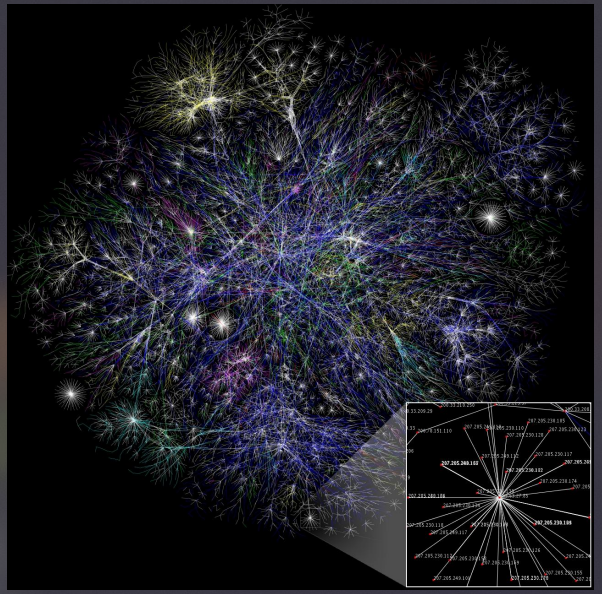
Traveling

Traveling

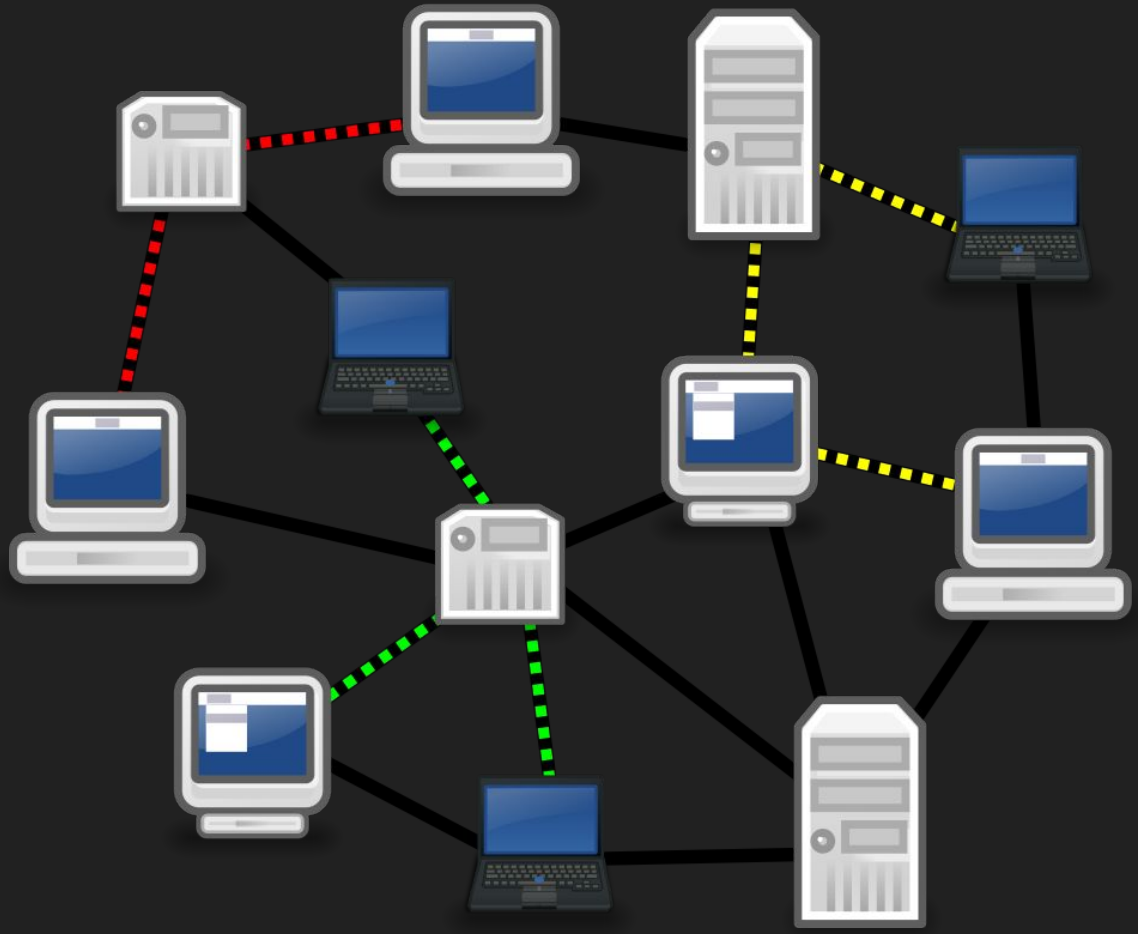
Traveling the Network



Programming the Network




RedBoot(tm) bootstrap and debug environment [RAM]
(Panasonic Avionics Corporation) release, version ("560328-212" v "1.07" b "0126
") - built 15:35:59, May 22 2013
Platform: SM-02 (I386)
Copyright (C) 2000, 2001, 2002, Red Hat, Inc.
RAM: 0x00000000-0x000a0000, 0x00100000-0x01000000 available
Current Boot Count is 0
verifying MBR... Fix MBR:
Partition 0: already exists
Partition 1: already exists
Partition 2: already exists
Partition 3: already exists
verifying image... OK.
== Executing kernel in 5 seconds - enter ^C to abort
Load Address 0x00000000
Image length 0x00e2f5d5
Loading kernel binary...
Read image signature... 1f 8b
Decompressing image...



Programming the Network

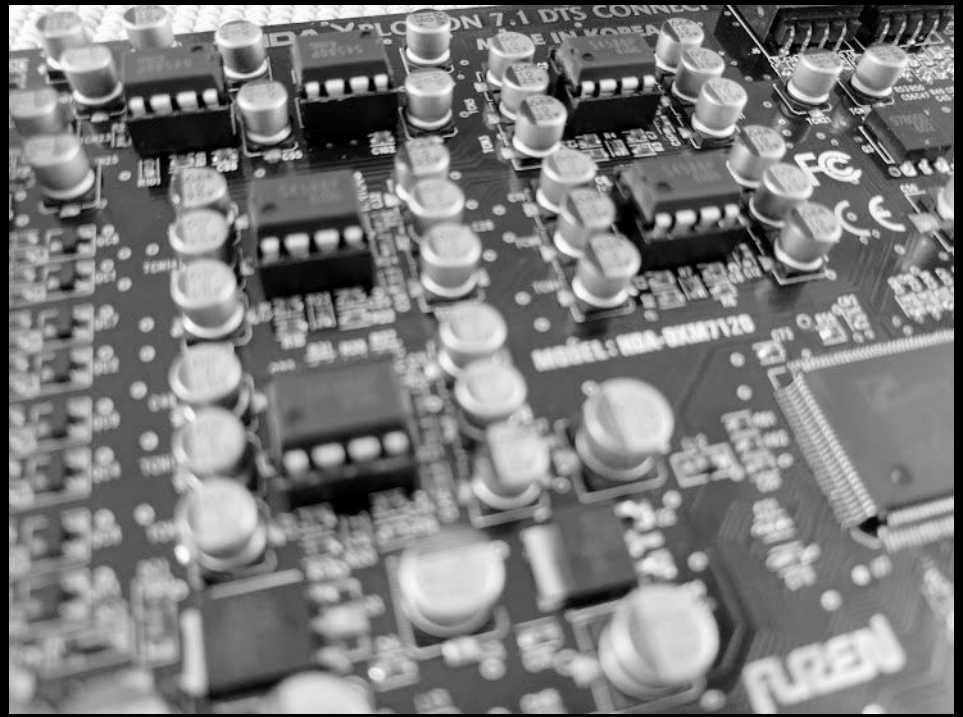
"There is no simultaneity at a distance."

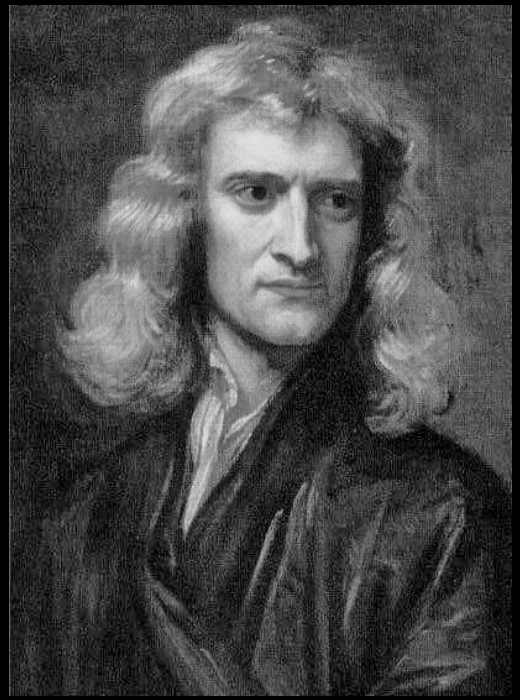
-- Pat Helland (2005)

Pat Helland

"Data on the Inside vs. Data on the Outside, Helland (2005) http://cidrdb.org/cidr2005/papers/P12.pdf

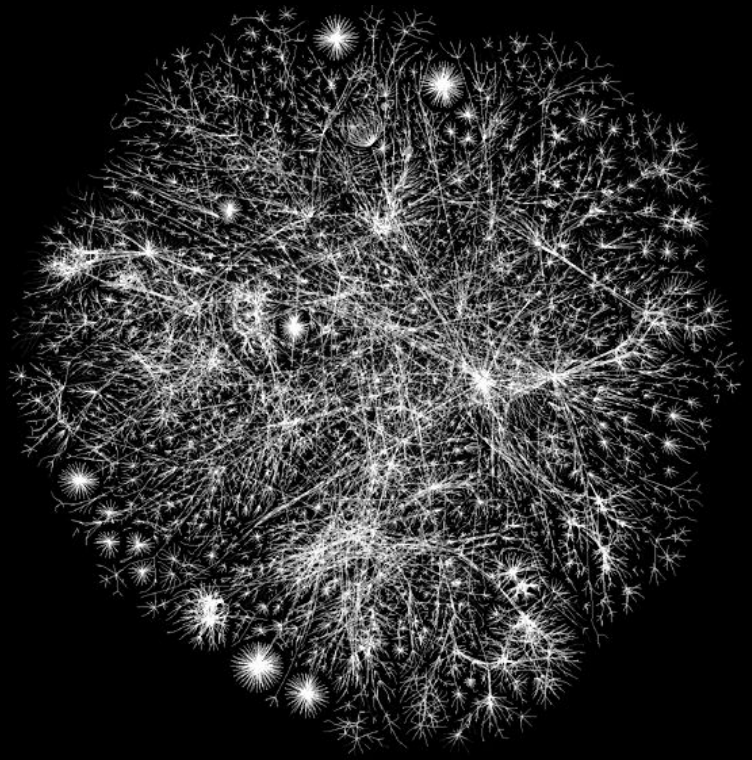
Newton rules the "inside"

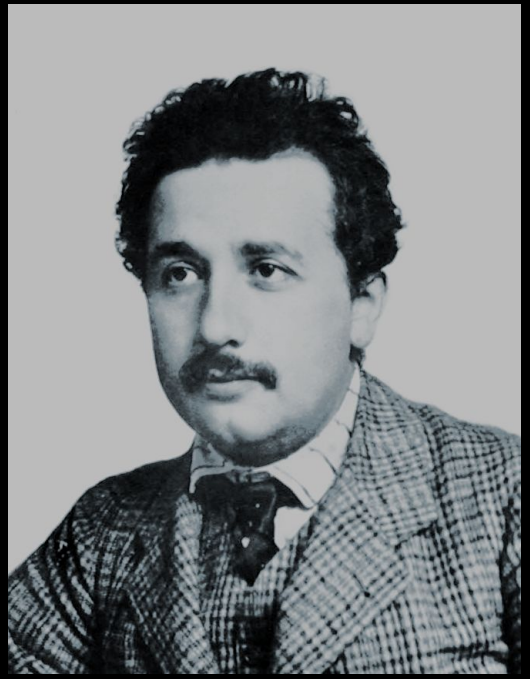




Sir Isaac Newton

Einstein rules the "outside"





Albert Einstein

Programming the Network

There is no simultaneity at a distance!
-- Similar to the speed of light bounding information
-- By the time you see a distant object, it may have changed!
-- By the time you see a message, the data may have changed!

Pat Helland

"Data on the Inside vs. Data on the Outside, Helland (2005) http://cidrdb.org/cidr2005/papers/P12.pdf

Programming the Network

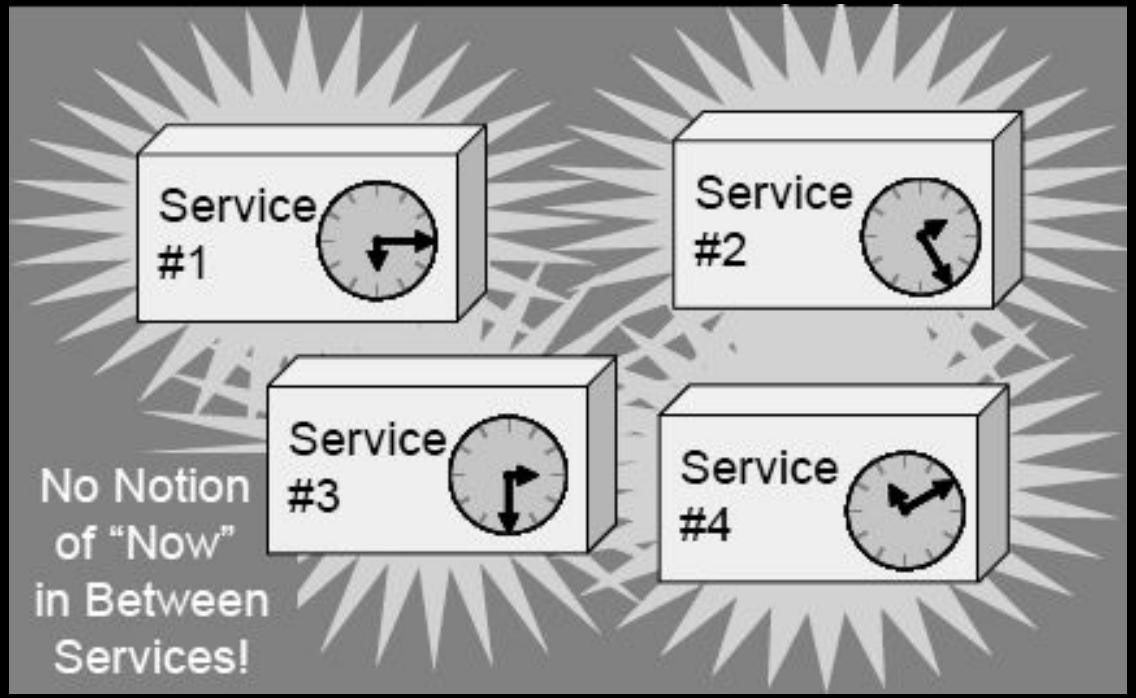
There is no simultaneity at a distance!
-- Similar to the speed of light bounding information
-- By the time you see a distant object, it may have changed!
-- By the time you see a message, the data may have changed!

Services, transactions, and locks bound simultaneity!
-- Inside a transaction, things are simultaneous
-- Simultaneity exists only inside a transaction!
-- Simultaneity exists only inside a service!

Pat Helland

"Data on the Inside vs. Data on the Outside, Helland (2005) http://cidrdb.org/cidr2005/papers/P12.pdf

Programming the Network



Pat Helland

"Data on the Inside vs. Data on the Outside, Helland (2005) http://cidrdb.org/cidr2005/papers/P12.pdf

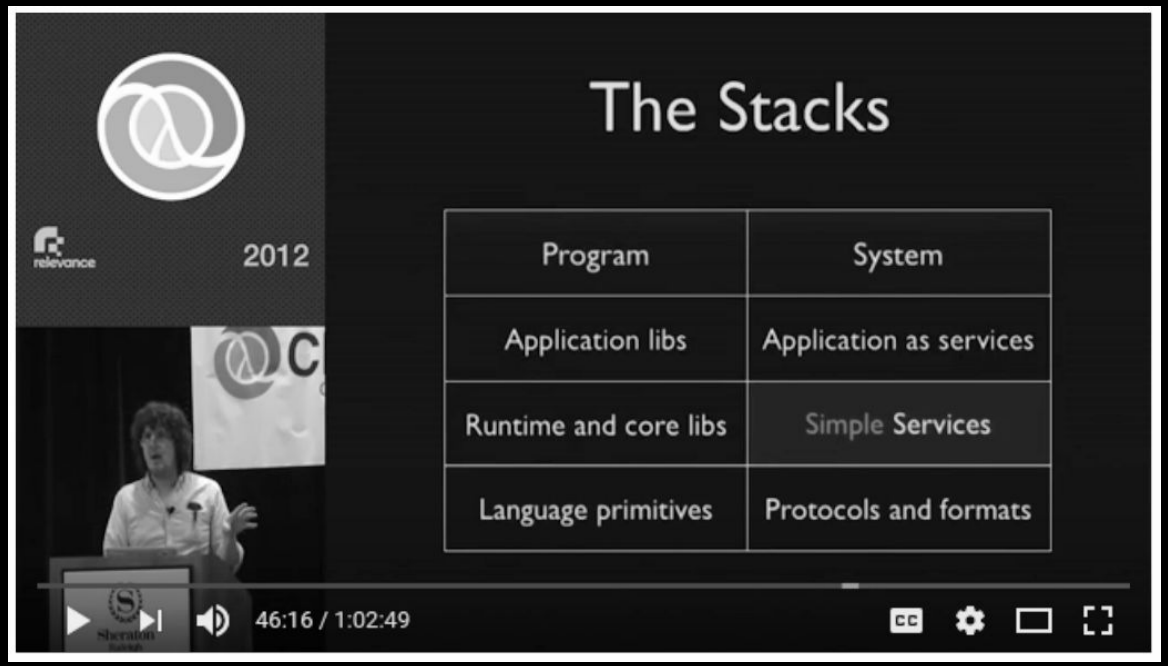
Fallacies of Distributed Computing (1994)

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

L Peter Deutsch

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

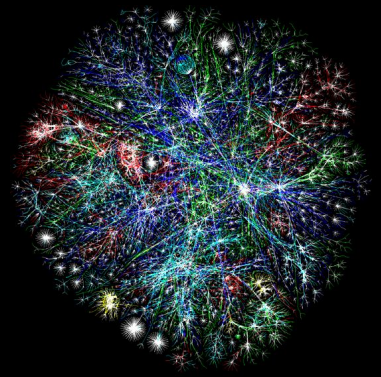
The Language of the System (2012)



Rich Hickey

The Language of the System, Rich Hickey (2012) https://www.youtube.com/watch?v=ROor6_NGIWU

Programming the Network brings new challenges



Microservices

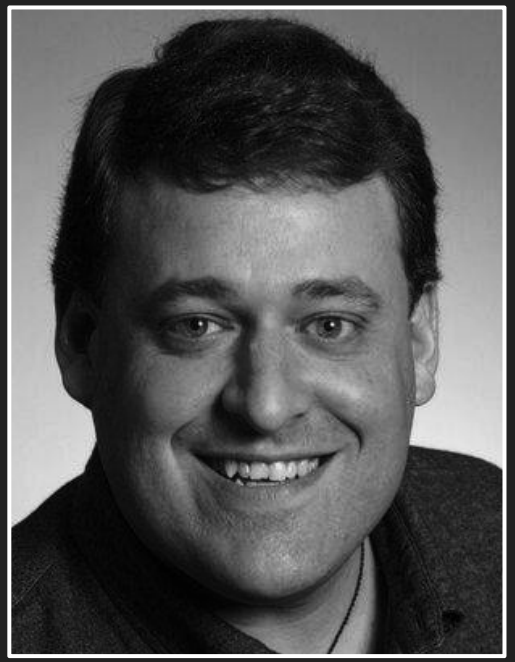
"An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms."

-- Martin Fowler, 2014

https://www.thoughtworks.com/insights/blog/microservices-nutshell

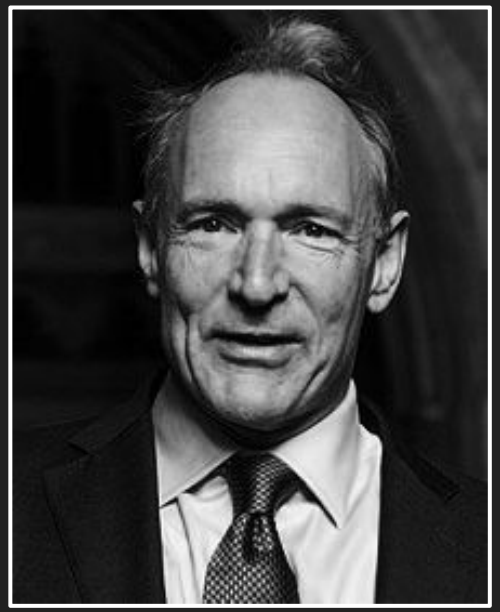
"Emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components."

-- Roy Fielding, 2000



"A universal linked information system, in which generality and portability are [most] important."

-- Tim Berners-Lee, 1989



Microservice Characteristics

- Make each program to one thing well
- Expect the output of every program to be the input of another program
- Design and build software to be tried early
- Use tools to lighten the programming task

Unix Operating Principles (1978)

- Make each program to one thing well
- Expect the output of every program to be the input of another program
- Design and build software to be tried early
- Use tools to lighten the programming task

https://en.wikipedia.org/wiki/Unix_philosophy

Loosely-coupled components running in an engineered system.

Three Types of Microservices

Three Types of Microservices

- Stateless
- Persistence
- Aggregator

Stateless Microservices

Stateless Microservices

- Simple processors (converters, translators, etc.)
- No dependence on other microservices
- No local data storage (disk I/O)

The most common MSC example, but the least useful!

Stateless Microservices

- No shared state
- Easy to replace
- Easy to scale up

Ephemeral Computing

Stateless Microservices

```
// http server handling data conversions
function conversionServer(request, response) {
  response = convertValue(request);
  return response;
}
```

WARNING: NOT REAL CODE!

Persistence Microservices

Persistence Microservices

- Simple (local) storage (reads and/or writes)
- Disk I/O dependent
- Possibly VM or one-U dependent

Commonly needed MSC, not the easiest to implement.

Persistence Microservices

- System of Record/Source of Truth
- Relatively easy to scale for reads (CQRS)
- No cross-service two-phase commits (Saga)

Durable Storage

Persistence Microservices

```
function updateOrders(request, response) {
  response = localStorage.write(request);
  return response;
}
```

WARNING: NOT REAL CODE!

Aggregator Microservices

Aggregator Microservices

- Depends on other ("distant") microservices
- Network dependent
- Usually Disk I/O dependence, too

The most often-needed; most challenging, too.

Aggregator Microservices

- Sequence vs. Parallel calls
- Timing is everything
- Easy to scale (should be…)

Workflow Choreography

Aggregator Microservices

```
function writeOrders(request, response) {
  var resourceList = ["customerDB", "orderDB", "salesDB"]'
  var serviceList = gatherResources(resourceList);
  response = serviceList(request)

  return response;
}
```

WARNING: NOT REAL CODE!

Three Types of Microservices

- Stateless (ephemeral)
- Persistence (durable)
- Aggregator (workflow)

But, what about the network?

Nygard's Stability Patterns

“Bugs will happen. They cannot be eliminated, so they must be survived instead.”

-- Michael T. Nygard


The Pragmatic Programmers
Release It!
Second Edition
Design and Deploy
Production-Ready Software
Michael T. Nygard
BETA

Nygard Stability Patterns

- **Timeout**
- **Circuit Breaker**
- **Bulkhead**
- **Steady State**
- **Fail Fast**
- **Handshaking**

"Nygard Stability Patterns" -- **Timeout**

"The timeout is a simple mechanism allowing you to stop waiting for an answer once you think it will not come."

-- Chapter 5.1

"Nygard Stability Patterns" -- **Timeout**

"The timeout is a simple mechanism allowing you to stop waiting for an answer once you think it will not come."

-- Ch 5.1

```
// set up proper shutdown
process.on('SIGTERM', function () {
  discovery.unregister(null, function(response) {
    try {
      uuidGenerator.close(function() {
      console.log('gracefully shutting down');
        process.exit(0);
      });
    } catch(e){}
  });
  setTimeout(function() {
    console.error('forcefully shutting down');
    process.exit(1);
  }, 10000);
});
```

WARNING: NOT REAL CODE!

"Nygard Stability Patterns" -- **Circuit Breaker**

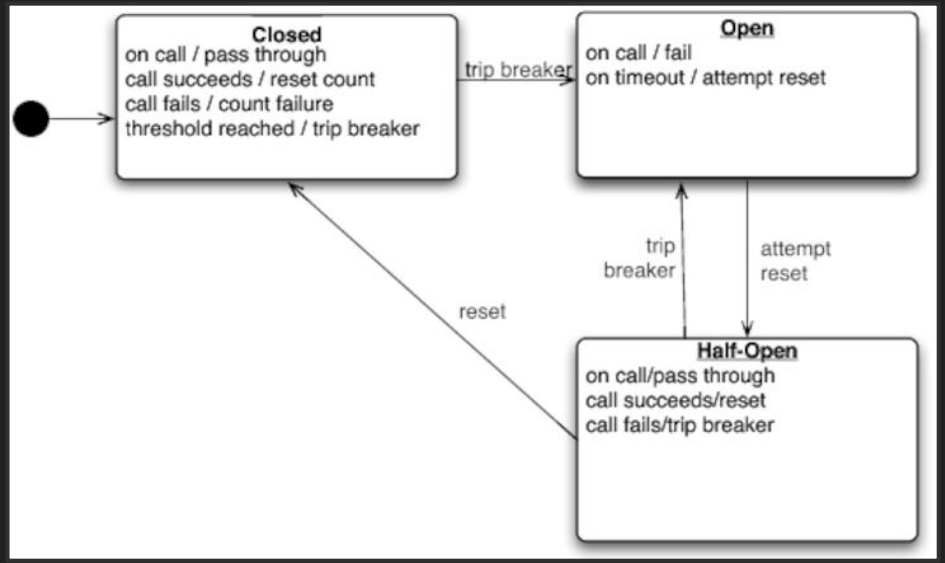
"Circuit breakers are a way to automatically degrade functionality when the system is under stress."

-- Chapter 5.2

"Nygard Stability Patterns" -- **Circuit Breaker**

"Circuit breakers are a way to automatically degrade functionality when the system is under stress."

-- Chapter 5.2



"Nygard Stability Patterns" -- **Bulkhead**

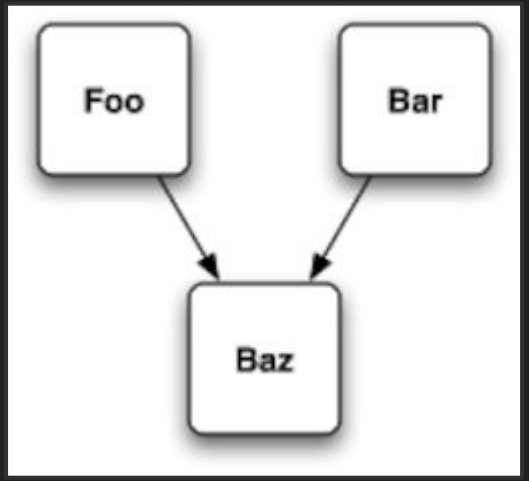
"The bulkhead enforces a principle of damage containment."

-- Chapter 5.3

"Nygard Stability Patterns" -- **Bulkhead**

"The bulkhead enforces a principle of damage containment."

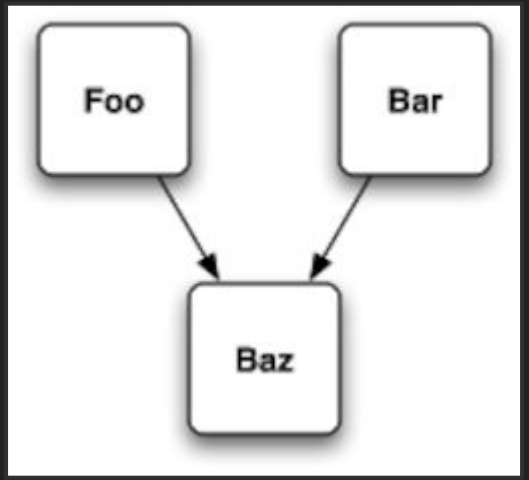
-- Chapter 5.3

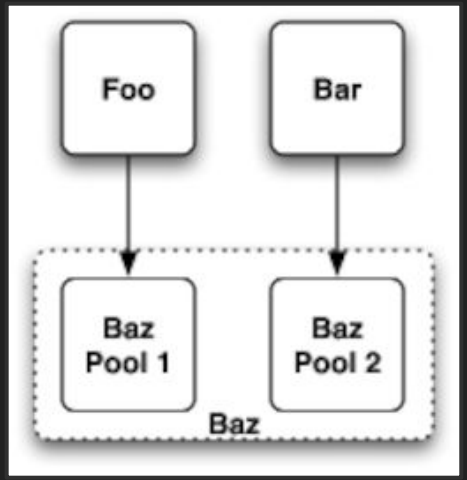


"Nygard Stability Patterns" -- **Bulkhead**

"The bulkhead enforces a principle of damage containment."

-- Chapter 5.3





"Nygard Stability Patterns" -- **Steady State**

"The system should be able to run indefinitely without human intervention."

-- Chapter 5.4

- Avoid fiddling
- Purge data w/ app logic
- Limit caching
- Roll the logs

"Nygard Stability Patterns" -- **Steady State**

"The system should be able to run indefinitely without human intervention."

-- Chapter 5.4

- Avoid fiddling
- Purge data w/ app logic
- Limit caching
- Roll the logs

"Nygard Stability Patterns" -- **Fail Fast**

"If the system can determine in advance that it will fail at an operation, it’s always better to fail fast."

-- Chapter 5.5

"Nygard Stability Patterns" -- **Fail Fast**

"If the system can determine in advance that it will fail at an operation, it's always better to fail fast."

-- Chapter 5.5

```
function bookOrders(orderList, timeBudget) {
  var status = false;
  var resources = ["customerdata","orderdata","salesdata"];
  setTimeout(function(resources) {
    var status = confirmResourceAvailability(resources);
    if(status===true && timeBudget>500) {
      try {
        status = writeOrders(orderList,resources);
      }
      catch (ex) {
        error("failed to write orders : {errordcode}",ex);
      }
    }
    else {
      error("failed to acquire resources : FAILFAST");
    }
  }, timeBudget);
}
```

WARNING: NOT REAL CODE!

"Nygard Stability Patterns" -- **Handshaking**

"Handshaking is all about letting the server protect itself by throttling its own workload."

-- Chapter 5.6

"Nygard Stability Patterns" -- **Handshaking**

"Handshaking is all about letting the server protect itself by throttling its own workload."

-- Chapter 5.6

```
function sendOrders(orderList, timeBudget) {
  if(
      (health.responseMS+health.latencyMS) < timeBudget
    ) {
      bookOrders.send(orderList,timeBudget);
    }
    else {
      error("failed to send orders: HEALTHCHECK");
    }
  }
```

WARNING: NOT REAL CODE!

"Nygard Stability Patterns" -- **Cache**

"Caching can reduce the load on the server and cut response times to a fraction of what they would be without caching."

-- Chapter 10.2

"Nygard Stability Patterns" -- **Cache**

"Caching can reduce the load on the server and cut response times to a fraction of what they would be without caching."

-- Chapter 10.2

```
// server marks data cache-able
function sendResults(response) {
  response.writeHead(status,
    { 'Content-Type' : 'text/plain',
      'Cache-Control': 'public,max-age=108000'}
  );
  response.end(value+'\n');
}
```

WARNING: NOT REAL CODE!

"Nygard Stability Patterns" -- **Cache**

"Caching can reduce the load on the server and cut response times to a fraction of what they would be without caching."

-- Chapter 10.2

```
// server marks data cache-able
function sendResults(response) {
  response.writeHead(status,
    { 'Content-Type' : 'text/plain',
      'Cache-Control': 'public,max-ag
  );
  response.end(value+'\n');
}
```

```
// client manages local cache
function getData(URL) {
  data = null;
  data = cache.read(URL);
  if(!data) {
    data = requestResults(URL);
    cache.write(URL,data);
  }
  return data;
}
```

WARNING: NOT REAL CODE!

Stabilizing Stateless Microservices

Stateless Microservices

```
// http server handling data conversions
function conversionServer(request, response) {
  response = convertValue(request);
  return response;
}
```

WARNING: NOT REAL CODE!

Networked Stateless

- *What if the work takes too long?*

Stable Stateless Microservices

```
// http server handling data conversions
function conversionServer(request, response) {
  if(request.timeBudget > my.averageResponse) {
    response = FailFastError(request);
  }
  else {
    response = convertValue(request);
  }
  return response;
}
```

1. Fail-Fast

WARNING: NOT REAL CODE!

Stabilizing Persistence Microservices

Persistence Microservices

```
function updateOrders(request, response) {
  response = localStorage.write(request);
  return response;
}
```

WARNING: NOT REAL CODE!

Networked Persistence

- *What if the work takes too long?*
- *What is the dependent service doesn't respond in time?*
- *What if the dependent service is down?*
- *What if the storage overflows (data, logs, etc.)?*

Stable Persistence Microservices

```
function updateOrders(request, response) {
  if(request.timeBudget < localStorage.latency) {
    response = FailFastError(request);
  }
  else {
    response = setTimeOut(circuitBreaker(
      localStorage.write(request),
      {timeout:10,maxFail:3,reset:30}
    ), timeBudget);
  }
  return response;
}
```

1. Fail-Fast
2. Timeout
3. Circuit Breaker
4. Steady State

WARNING: NOT REAL CODE!

Stabilizing Aggregator Microservices

Aggregator Microservices

```
function writeOrders(request, response) {
  var resourceList = ["customerDB", "orderDB", "salesDB"]'
  var serviceList = gatherResources(resourceList);
  response = serviceList(request)

  return response;
}
```

WARNING: NOT REAL CODE!

Networked Aggregators

- *What if the work takes too long?*
- *What if a dependent services doesn't respond in time?*
- *What if a dependent service is down?*
- *What if storage overflows (data, logs, etc.)?*
- *What if a dependent service is unhealthy?*
- *What if traffic for a service spikes?*

Stable Aggregator Microservices

```
function writeOrders(request, response) {
  var resourceList = ["customerDB", "orderDB", "salesDB"]'

  setTimeOut(function(request,response, resourceList) {
    var serviceList = gatherResources(resourceList);
    if(serviceList.estimatedCost > request.timeBudget) {
      response = FailFast(request);
    }
    else {
      if(serviceList.healthy === true) {
        circuitBreaker(serviceList, request,
        {timeout:10,maxFail:3,reset:30});
      }
    }
  },request.timeBudget);

  return response;
}
```

WARNING: NOT REAL CODE!

1. Fail-Fast
2. Timeout
3. Circuit Breaker
4. Steady State
5. Handshaking
6. Bulkhead

Nygard's Admonition...

Joe asks:

Is All This Clutter Really Necessary?

You may think, as I did when porting the sockets library, that handling all the possible timeouts creates undue complexity in your code. It certainly adds complexity. You may find that half your code is devoted to error handling instead of providing features. I argue, however, that the essence of aiming for production—instead of aiming for QA—is handling the slings and arrows of outrageous fortune. That error-handling code, if done well, adds resilience. Your users may not thank you for it, because nobody notices when a system *doesn't* go down, but you will sleep better at night.

Applying Nygard's Patterns to Services

- **Stateless**
  - ***fail fast***
- **Persistence**
  - ***fail fast, timeout, circuit breaker, steady state***
- **Aggregation**
  - ***fail fast, timeout, circuit breaker, steady state, handshaking, bulkhead***

Apply Nygard's Stability Patterns
to improve the health
of your components and your system.


BUT WAIT,
there's more!

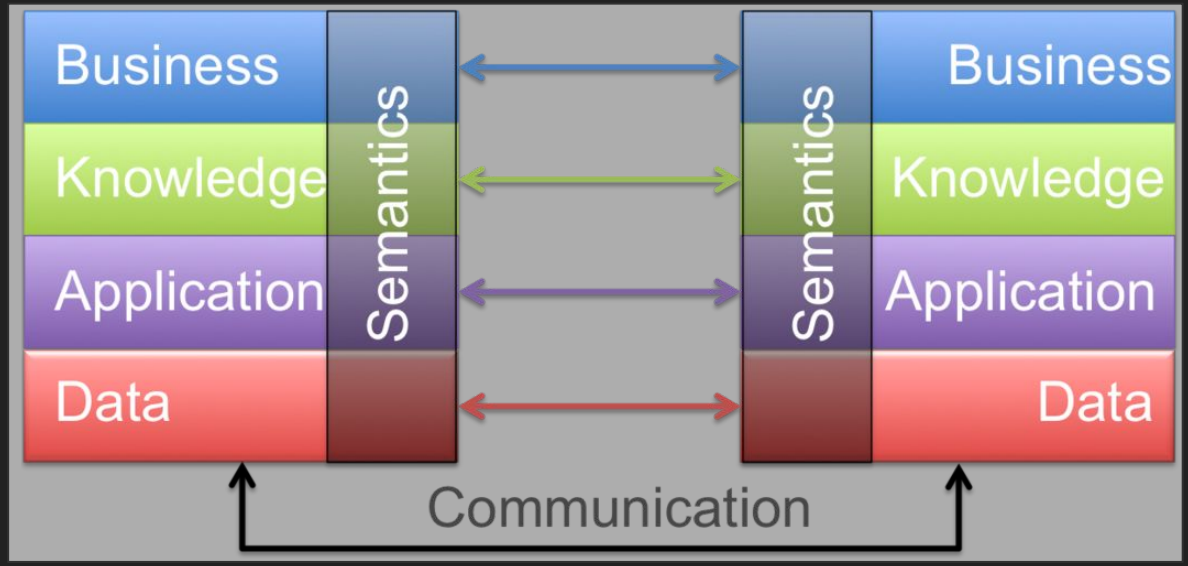
Aim for Interop, not Integration...

"Interoperation is peer to peer. Integration is where a system is subsumed within another."

-- Michael Platt, Microsoft

https://blogs.technet.microsoft.com/michael_platt/2005/08/30/integration-and-interoperability/

Aim for Interop, not Integration...



Include time/distance in your models

"There is no simultaneity at a distance."

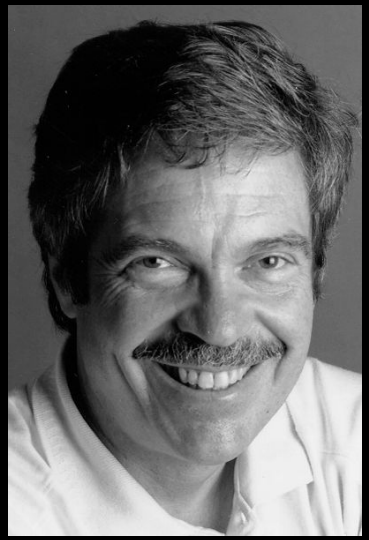
-- Pat Helland, Salesforce

Pat Helland

https://blogs.technet.microsoft.com/michael_platt/2005/08/30/integration-and-interoperability/

Include time/distance in your models

"I'm sorry that coined the term 'objects' for this topic. The big idea is 'messaging'."



Alan Kay, 1998

Remember, you're programming the *network*

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

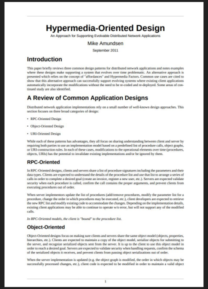
L Peter Deutsch

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

Remember, you're programming the *network*

- **Safety**

https://www.w3.org/2011/10/integration-workshop/p/hypermedia-oriented-design.pdf



Hypermedia-Oriented Design

An Approach for Supporting Evolvable Distributed Network Applications

Mike Amundsen

September 2011

Introduction

This paper briefly reviews three common design patterns for distributed network applications and notes examples where these designs make supporting a system that evolves over time problematic. An alternative approach is presented which relies on the concept of "affordances" and Hypermedia Factors. Common use cases are cited to show that this alternative approach can successfully support evolving systems where existing client applications automatically incorporate the modifications without the need to be re-coded and re-deployed. Some areas of continued study are also identified.

A Review of Common Application Designs

Distributed network application implementations rely on a small number of well-known design approaches. This section focuses on three broad categories of design:

- RPC-Oriented Design
- Object-Oriented Design
- URI-Oriented Design

While each of these patterns has advantages, they all focus on sharing understanding between client and server by requiring both parties to use an implementation model based on a predefined list of procedure calls, object graphs, or URI-construction rules. In each of these cases, modifications to the operational elements over time (procedures, objects, URIs) has the potential to invalidate existing implementations and/or be ignored by them.

RPC-Oriented

In RPC-Oriented designs, clients and servers share a list of procedure signatures including the parameters and their data types. Clients are expected to understand the details of the procedure list and use that list to arrange a series of calls in order to complete a desired goal (create an invoice, register a new user, etc.). Servers are expected validate security when each procedure is called, confirm the call contains the proper arguments, and prevent clients from executing procedures out of order.

When server implementors update the list of procedures (add/remove procedures, modify the parameter list for a procedure, change the order in which procedures may be executed, etc.), client developers are expected to retrieve the new RPC list and modify existing code to accommodate the changes. Depending on the implementation details, existing client applications may be able to continue to operate w/o error, but will not support any of the modified calls.

In RPC-Oriented models, the client is "bound" to the procedure list.

Object-Oriented

Object-Oriented designs focus on making sure clients and servers share the same object model (objects, properties, hierarchies, etc.). Clients are expected to maintain a copy of the object model, serialize objects for submitting to the server, and recognize serialized objects sent from the server. It is up to the client to use this object model in order to reach a desired goal. Servers are expected to validate security when handling requests, confirm the schema of the serialized objects it receives, and prevent clients from passing object serializations out of order.

When the server implementation is updated (e.g. the object graph is modified, the order in which objects may be successfully processed changes, etc.), client code is expected to be modified in order to maintain a valid object

1

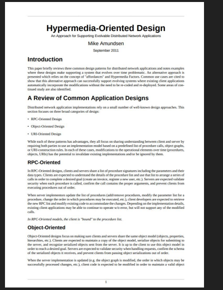
Remember, you're programming the *network*

- **Safety**

The HTTP protocol supports a number of "safe" actions such as HEAD, and GET.

The HTTP methods PUT, POST, and DELETE are categorized as "unsafe" actions.

https://www.w3.org/2011/10/integration-workshop/p/hypermedia-oriented-design.pdf


Hypermedia-Oriented Design
An Approach for Supporting Evolvable Distributed Network Applications
Mike Amundsen
September 2011

Introduction

A Review of Common Application Designs

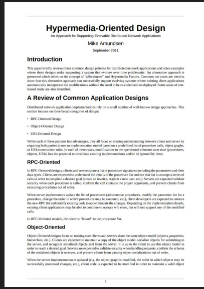
RPC-Oriented

Object-Oriented

Remember, you're programming the *network*

- Safety
- **Idempotence**

https://www.w3.org/2011/10/integration-workshop/p/hypermedia-oriented-design.pdf



Hypermedia-Oriented Design

An Approach for Supporting Evolvable Distributed Network Applications

Mike Amundsen

September 2011

Introduction

This paper briefly reviews three common design patterns for distributed network applications and notes examples where these designs make supporting a system that evolves over time problematic. An alternative approach is presented which relies on the concept of "affordances" and Hypermedia Factors. Common use cases are cited to show that this alternative approach can successfully support evolving systems where existing client applications automatically incorporate the modifications without the need to be re-coded and re-deployed. Some areas of continued study are also identified.

A Review of Common Application Designs

Distributed network application implementations rely on a small number of well-known design approaches. This section focuses on three broad categories of design:

- RPC-Oriented Design
- Object-Oriented Design
- URI-Oriented Design

While each of these patterns has advantages, they all focus on sharing understanding between client and server by requiring both parties to use an implementation model based on a predefined list of procedure calls, object graphs, or URI-construction rules. In each of these cases, modifications to the operational elements over time (procedures, objects, URIs) has the potential to invalidate existing implementations and/or be ignored by them.

RPC-Oriented

In RPC-Oriented designs, clients and servers share a list of procedure signatures including the parameters and their data types. Clients are expected to understand the details of the procedure list and use that list to arrange a series of calls in order to complete a desired goal (create an invoice, register a new user, etc.). Servers are expected validate security when each procedure is called, confirm the call contains the proper arguments, and prevent clients from executing procedures out of order.

When server implementors update the list of procedures (add/remove procedures, modify the parameter list for a procedure, change the order in which procedures may be executed, etc.), client developers are expected to retrieve the new RPC list and modify existing code to accommodate the changes. Depending on the implementation details, existing client applications may be able to continue to operate w/o error, but will not support any of the modified calls.

In RPC-Oriented models, the client is "bound" to the procedure list.

Object-Oriented

Object-Oriented designs focus on making sure clients and servers share the same object model (objects, properties, hierarchies, etc.). Clients are expected to maintain a copy of the object model, serialize objects for submitting to the server, and recognize serialized objects sent from the server. It is up to the client to use this object model in order to reach a desired goal. Servers are expected to validate security when handling requests, confirm the schema of the serialized objects it receives, and prevent clients from passing object serializations out of order.

When the server implementation is updated (e.g. the object graph is modified, the order in which objects may be successfully processed changes, etc.), client code is expected to be modified in order to maintain a valid object

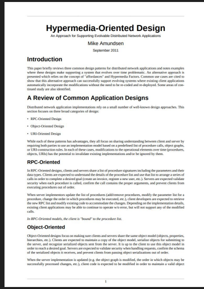
1

Remember, you're programming the *network*

- Safety
- **Idempotence**

In HTML when a FORM element has the METHOD property set to "get" this represents an idempotent action.

When the same property is set to "post" the affordance represents a non-idempotent action.



Hypermedia-Oriented Design

An Approach for Supporting Evolvable Distributed Network Applications

Mike Amundsen

September 2011

Introduction

This paper briefly reviews three common design patterns for distributed network applications and notes examples where these designs make supporting a system that evolves over time problematic. An alternative approach is presented which relies on the concept of "affordances" and Hypermedia Factors. Common use cases are cited to show that this alternative approach can successfully support evolving systems where existing client applications automatically incorporate the modifications without the need to be re-coded and re-deployed. Some areas of continued study are also identified.

A Review of Common Application Designs

Distributed network application implementations rely on a small number of well-known design approaches. This section focuses on three broad categories of design:

- RPC-Oriented Design
- Object-Oriented Design
- URI-Oriented Design

While each of these patterns has advantages, they all focus on sharing understanding between client and server by requiring both parties to use an implementation model based on a predefined list of procedure calls, object graphs, or URI-construction rules. In each of these cases, modifications to the operational elements over time (procedures, objects, URIs) has the potential to invalidate existing implementations and/or be ignored by them.

RPC-Oriented

In RPC-Oriented designs, clients and servers share a list of procedure signatures including the parameters and their data types. Clients are expected to understand the details of the procedure list and use that list to arrange a series of calls in order to complete a desired goal (create an invoice, register a new user, etc.). Servers are expected validate security when each procedure is called, confirm the call contains the proper arguments, and prevent clients from executing procedures out of order.

When server implementors update the list of procedures (add/remove procedures, modify the parameter list for a procedure, change the order in which procedures may be executed, etc.), client developers are expected to retrieve the new RPC list and modify existing code to accommodate the changes. Depending on the implementation details, existing client applications may be able to continue to operate w/o error, but will not support any of the modified calls.

In RPC-Oriented models, the client is "bound" to the procedure list.

Object-Oriented

Object-Oriented designs focus on making sure clients and servers share the same object model (objects, properties, hierarchies, etc.). Clients are expected to maintain a copy of the object model, serialize objects for submitting to the server, and recognize serialized objects sent from the server. It is up to the client to use this object model in order to reach a desired goal. Servers are expected to validate security when handling requests, confirm the schema of the serialized objects it receives, and prevent clients from passing object serializations out of order.

When the server implementation is updated (e.g. the object graph is modified, the order in which objects may be successfully processed changes, etc.), client code is expected to be modified in order to maintain a valid object

1

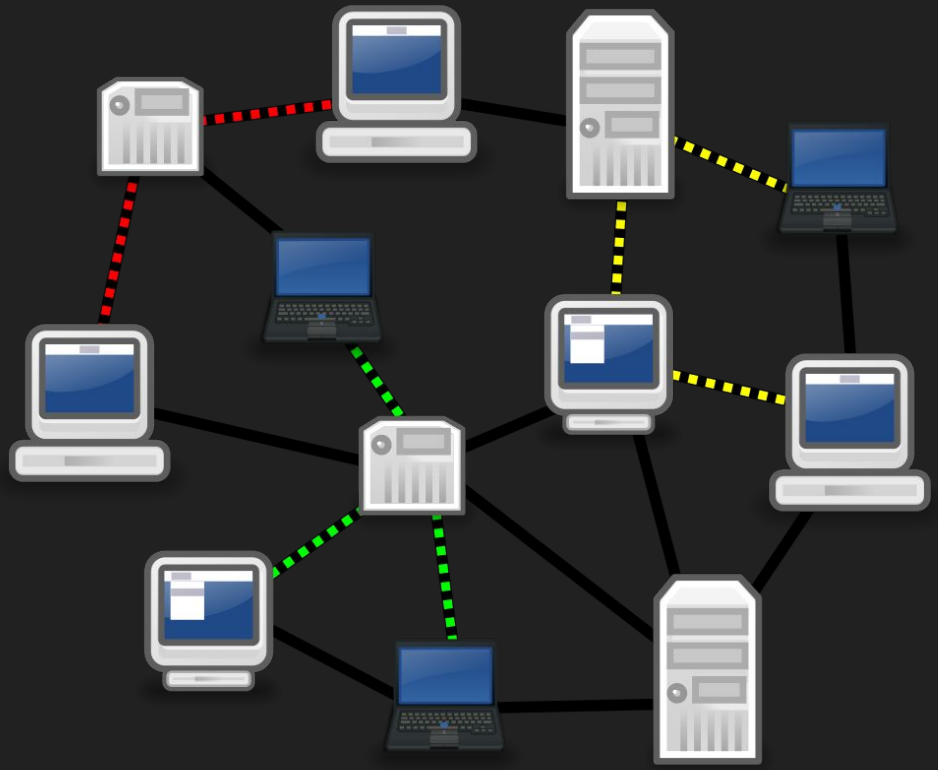
https://www.w3.org/2011/10/integration-workshop/p/hypermedia-oriented-design.pdf

Other Considerations...

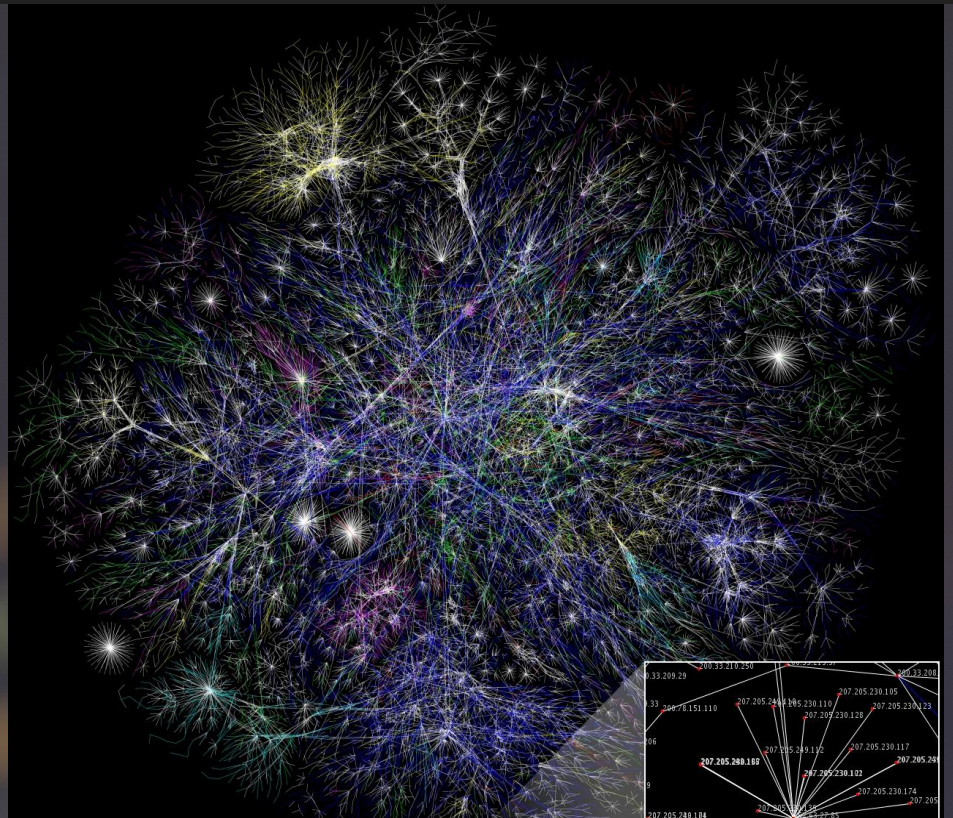
- Interop vs. Integration
- Time & Distance
- Safety & Idempotence

So...

We need microservices...



So that we can program the network...

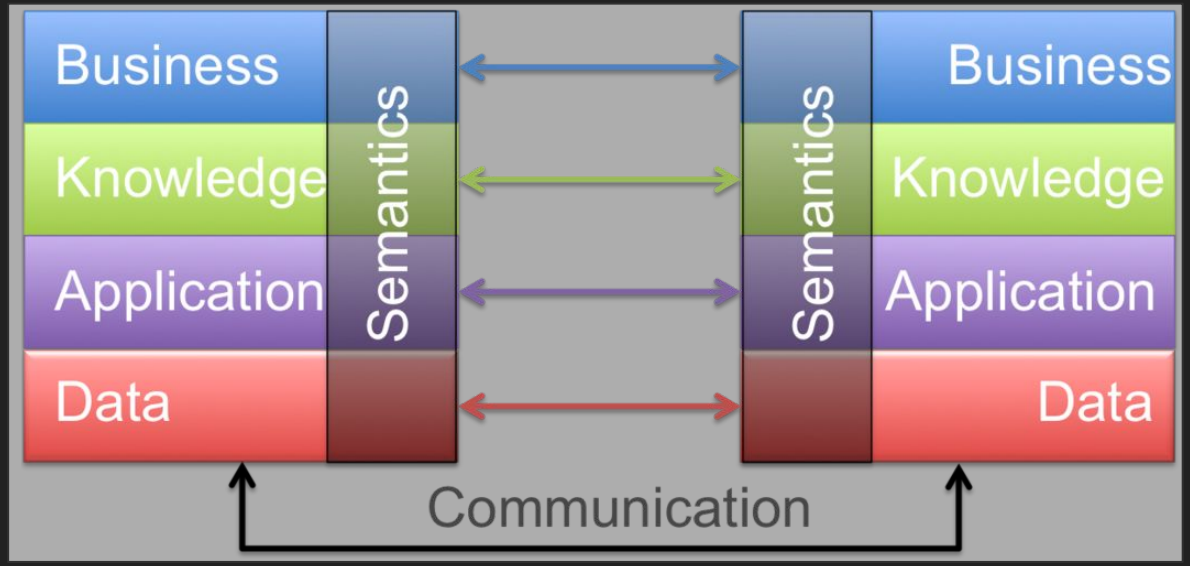


Which means applying patterns to our code..,

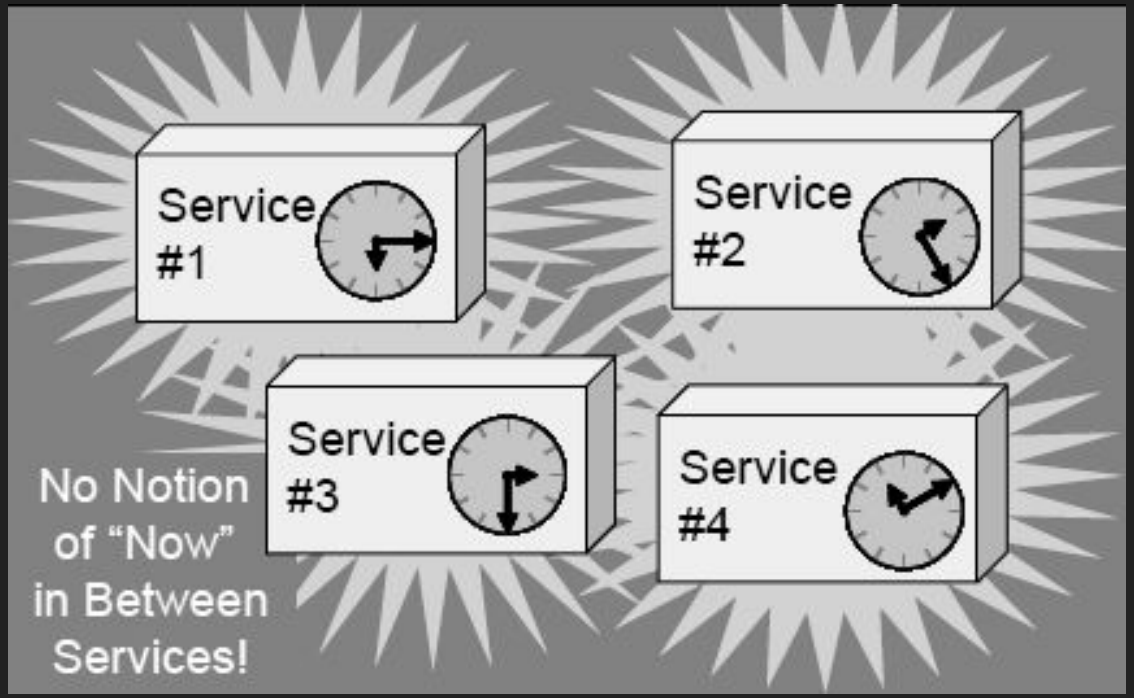
```
function writeOrders(request, response) {
  var resourceList = ["customerDB", "orderDB", "salesDB"]'

  setTimeOut(function(request,response, resourceList) {
    var serviceList = gatherResources(resourceList);
    if(serviceList.estimatedCost > request.timeBudget) {
      response = FailFast(request);
    }
    else {
      if(serviceList.healthy === true) {
        circuitBreaker(serviceList, request,
        {timeout:10,maxFail:3,reset:30});
      }
    }
  },request.timeBudget);

  return response;
}
```

1. Fail-Fast
2. Timeout
3. Circuit Breaker
4. Steady State
5. Handshaking
6. Bulkhead

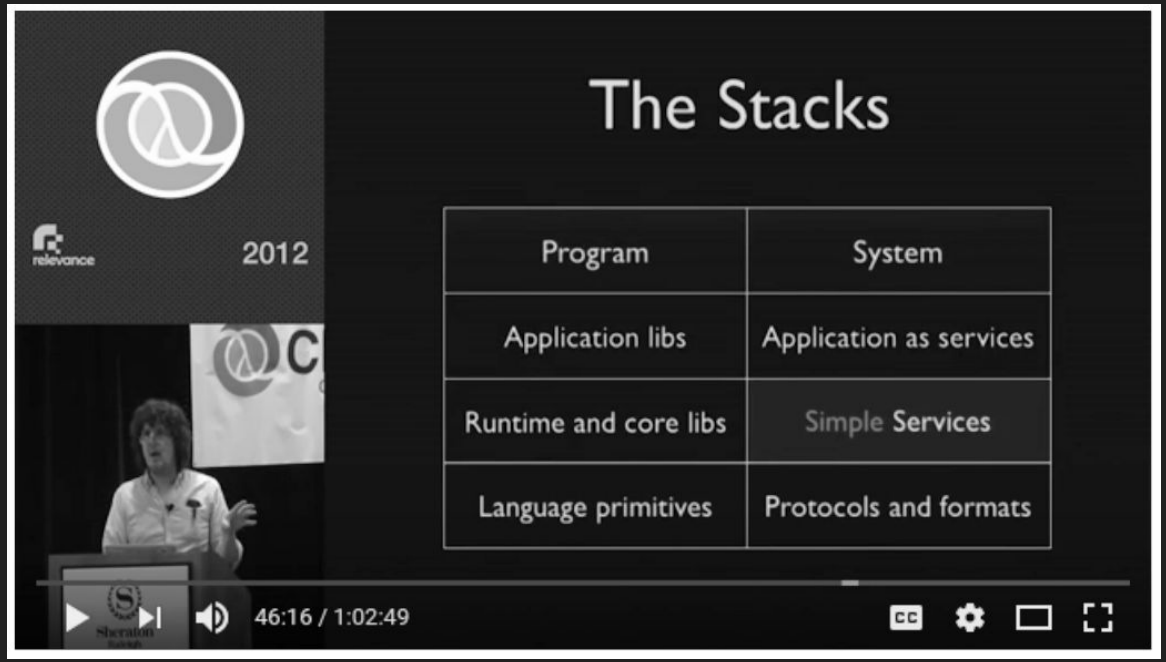
And that means understanding the role of semantics...



And the role of distance & time...



And constantly reminding ourselves of the challenge.



That's a lot!

The Best Software Architecture

"The best software architecture 'knows' what changes often and makes that easy."

- Paul Clements

Learning the Three Types of Microservices

Mike Amundsen
@mamund
API Academy