Lesson 9: Reactive Systems and the Actor Model

Phillip J. Windley, Ph.D.

CS462 – Large-Scale Distributed Systems

Contents

Introduction

The Internet of Things is upon us, or at least so the technology pundits tell us. But this first phase of the Internet of Things, where we are wowed by simply controlling previously unconnected devices with a mobile app, will be short lived. Ultimately it is bound to disappoint us since these systems are too centralized, hierarchical, and incompatible.

One major obstacle to building the Internet of Things we want is our style of programming. We build systems that are based on a Web 2.0-style architecture where your interactions with your devices are conceived of as services provided to you by some vendor.

The current style of programming the Internet of Things creates silos. These silos not only fragment the data belonging to a person, but also ensure that interactions between a device and the other things a person owns or uses must be mediated by relationships between the vendors.

The past several lessons have included concepts and techniques that let us conceive of a programming style that supports systems that are more decentralized, loosely coupled, and flexible. This lesson introduces a programming model that pulls those concepts together and provides a concrete foundation for peer-to-peer architectures.

Lesson Objectives

After completing this lesson, you should be able to:

1. Describe why and how the Internet of Things will change programming and distributed systems
2. Understand reactive programming and it’s goals
3. Explain and use the actor model as embodied in persistent compute objects
4. Understand and use event expressions to describe event scenarios
5. Describe and use common event processing and reactive programming patterns

Reading

Read the following:

- *The Live Web* by Phillip J. Windley, Chapters 1-4, 11 (textbook)
- On Hierarchies and Networks by Phillip J. Windley (http://www.windley.com/archives/2011/09/on_hierarchies_and_networks.shtml)
- What is Reactive Programming? by Kevin Webber (https://medium.com/reactive-programming/what-is-reactive-programming-bc9fa7f4a7fc)
- Reactive Programming with Picos by Phillip J. Windley, Read the article and the articles it links to (two levels deep). (http://www.windley.com/archives/2015/11/reactive_programming_with_picos.shtml)

Additional Resources

Additional Resources:

- *The Live Web* by Phillip J. Windley, Chapters 12-14 (textbook)
- *Reactive Messaging Patterns with the Actor Model*, by Vaughn Vernon.

The Internet of Things

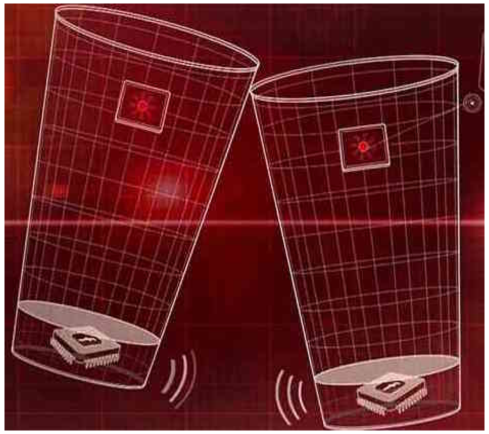
The current model of the Internet of Things can’t be expanded to successfully connect the trillions of things that make up our future computing environment

Buddy Cups

One of my favorite futuristic examples of how computers would be in everything was putting them in drinking glasses. Until Budweiser did it in 2013…

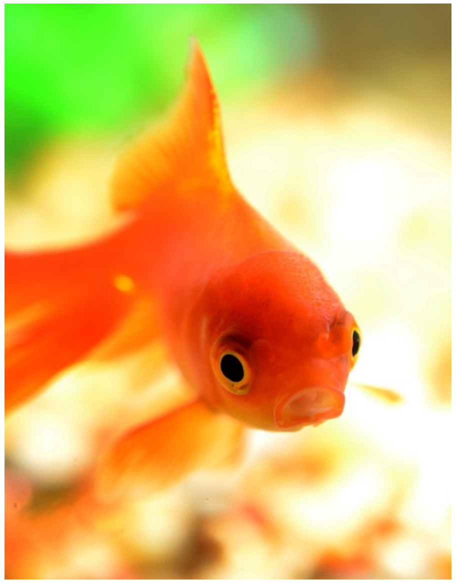
Budweiser's Buddy Cup* has a connected computer, accelerometer, and QR code. When you scan the QR code, an OAuth flow sets up a connection between the glass and your Facebook account. Whenever you clink glasses with someone, you become Facebook friends and exchange contact information.

These aren't cheap enough to give away…yet. But they will be and then computers will be in all your tablewear and everywhere else.



*http://www.engadget.com/2013/04/29/budweiser-buddy-cup-toast-for-facebook-friending/

Ambience

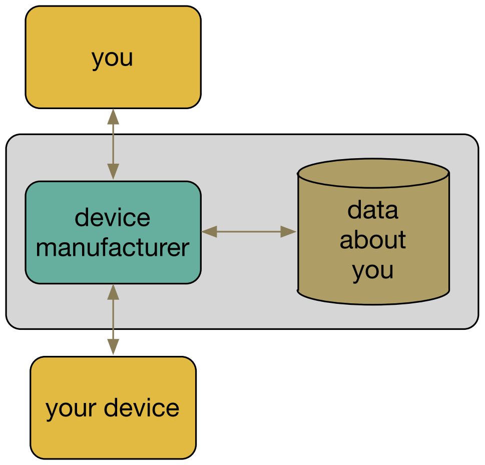


A fish doesn’t know it lives in water. Water is *ambient* from the perspective of the fish. Similarly, the Internet of Things will lead to computing that is immersive and pervasive. Computing will be ambient.

To date, most of our computers have been things we used, not something we lived in. When computers are in everything, they will not be things we use the same way we use an iPhone. Excepting perhaps configuration, our interactions with them won’t be through the traditional user interfaces we’ve become accustomed to.

Instead, our every interaction will be in some way mediated by computation. We will simple be immersed in computation.

The Current IoT Architecture



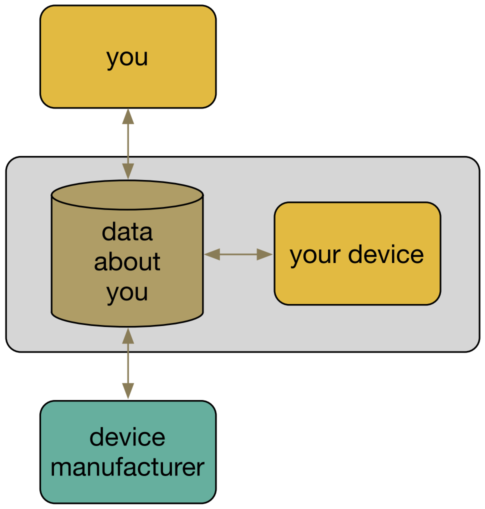
The architecture of this kind of ambient computing environment will be very different from today's networks.

Today's networks are meant to handle billions of devices and they suffer from an over-reliance on centralized directories, identities, and authorities. They have very little capability for creating dynamic collections of peers that come together to achieve some purpose.

The current Internet of Things is based on a Web 2.0 architectural model that focuses on the manufacturer of the device rather than the owner.

This architecture places the device manufacturer and their systems in between you and your device. What's more, the manufacturer stores, and controls, data about you.

An Person-Centric Architecture



Alternative architectures give the owner a direct relationship with their data, devices, and the systems that control them.

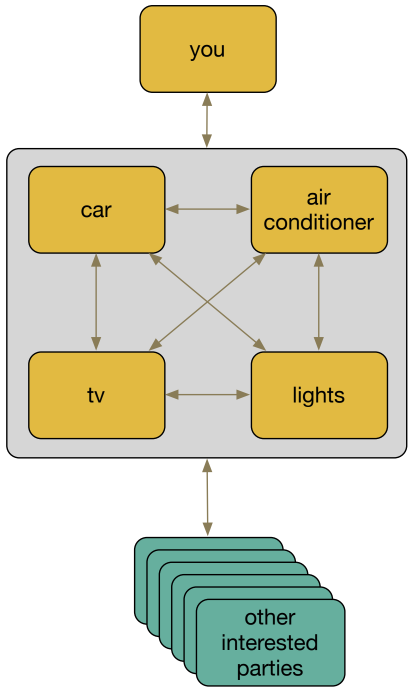
In this architecture, the owner is at the center, controlling the systems that their devices use and the data that those devices generate. The diagram has the same pieces, they're simply rearranged

The device manufacturer, and others, can also have access to the device and its data so long as the owner gives permission.

Imagine a connected-car, for example. The owner has the preeminent relationship with the vehicle. But the manufacturer, dealer, insurance company, finance company, service vendor would also want a relationship with the vehicle.

We can even imagine the vehicle having relationships with the roads it drives on, the garage it parks in, the traffic signals it passes, and even other vehicles on the road.

The Internet of My Things



This alternative architecture puts the owner in control, creating the Internet of My Things.

In the Internet of My Things, your devices are peers and don't rely on manufacturer-mediated interactions to work together. Rather, they communicate directly and are able to coordinate their actions.

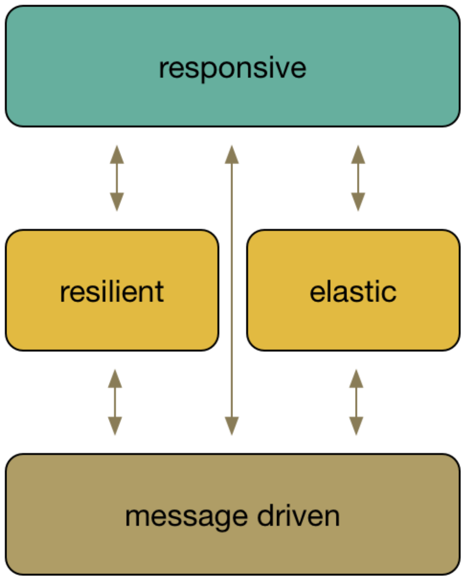
For this model to succeed, the underlying computation must be decentralized, heterarchical, and interoperable. Today's Web 2.0-inspired Internet of Things architectures can't provide these features.

Reactive programming, and the actor model in particular, have properties that mitigate some of the limitations that today's Internet of Things exhibits and allow us to more easily model the Internet of My Things.

Reactive Programming and the Actor Model

Reactive programming with the actor model offers significant advantages in modeling interaction on the Internet of Things

The Reactive Manifesto



The Reactive Manifesto describes a type of system architecture that has four characteristics (quoting from the manifesto):

- **Responsive:** The system responds promptly if at all possible.
- **Resilient:** The system stays responsive in the face of failure.
- **Elastic:** The system stays responsive under varying workload.
- **Message Driven:** Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages.

These are often represented as a stack since the only explicit architectural choice is to be message driven. The others are emergent properties of that and other architectural choices.

Reactive Systems

The goal of the reactive manifesto is to promote architectures and systems that are more flexible, loosely-coupled, and scalable while making them amenable to change.

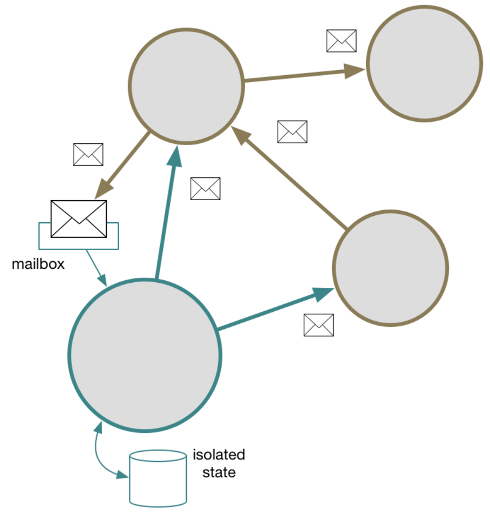
Reactive principles are best practices that can be applied to components at any level in the architecture. While many think of reactive systems being about user-interface design, the principles apply to the underlying systems as well.

Messaging is at the heart of a reactive system because messaging enables asynchronous interactions.

Messaging naturally provides important properties of a reactive system. Messaging enforces isolation since messages only share snapshots of a system’s state. Similarly, messages are easily routed over a network between peers in support of location transparency.

The manifesto doesn't specify a development methodology, so reactive systems can be built using a wide variety of systems, frameworks, and languages.

The Actor Model



Actors are a good way of building reactive systems. Actors are a distributed-programming model invented by Carl Hewitt in 1973.

Actors extend message-driven programming with two additional required properties. In response to a received message,

1. actors send messages to other actors,
2. actors create other actors, and
3. actors change their internal state.

Each actor has a mailbox and it's own state that is isolated from that of other actors. Message processing happens due to state changes that can affect the behavior of the next message.

Programming Actors

Actors are more than a theoretical model of concurrent computation. They are meant to provide practical blueprint for building distributed systems.

So long as the messaging system provides a way to address them, actors abstract away network boundaries.

The messaging system and state isolation also abstracts away implementation differences between actors. Provided actors understand the message, the details of their internal implementation is immaterial to other actors.

The messaging system in actors is simple: there are no guarantees on when, or whether, messages will be delivered.

Isolation and simple messaging mean that actor-based systems implement lock-free concurrency. Non-blocking interactions lead to loose coupling.

Because actors can create other actors, actor-based systems are incredibly dynamic and adaptive.

Persistent Compute Objects

Persistent compute objects are an actor-model-based programming system that aids in building computing structures that are decentralized, heterarchical and interoperable

Persistent Compute Objects

Persistent compute objects, or picos, implement the actor model of distributed computation.

Picos are:

- **event-driven**: They respond to events by changing state and sending new events to other picos.
- **persistent**: They exist from when they are created until they are explicitly deleted. Picos retain state based on past operations.
- **unique**: They have an identity that is immutable. While a pico’s state may change, its identity does not.
- **online**: They are continuously available on the Internet and respond to events and queries. Picos cannot crash.
- **concurrent**: They operate independently of one another and process events and queries asynchronously.
- **rule-based**: Their behavior is expressed as rules that pattern-match against incoming events. Put another way, rules listen for events on the pico's internal event bus.

Programming with Picos

Solving interesting programming problems with picos usually involves creating a network of picos that solve the problem together.

A pico-based application is a collection of picos operating together to achieve some purpose. A single pico is almost never very interesting.

Picos have channels that are analogous to the mailboxes in the actor model. Picos can send events to other picos over channels. In addition, picos can query each other over these same channels.

Programmers solve problems with picos by designing the interactions between them and then programming each pico to respond appropriately.

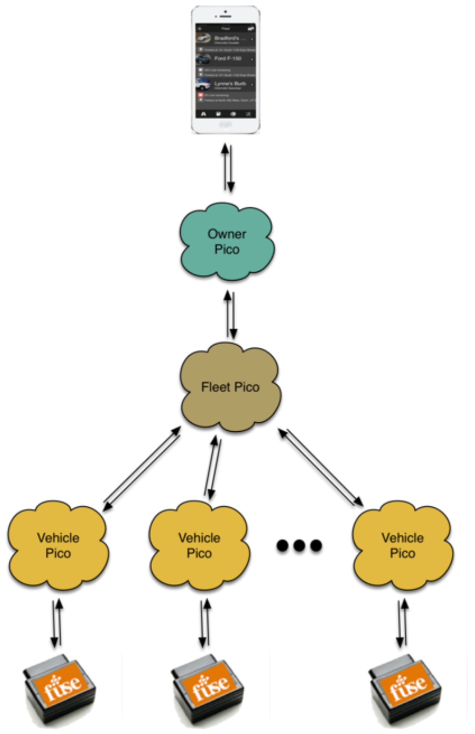
Pico Example: Fuse

Fuse is a connected-car system built with picos.

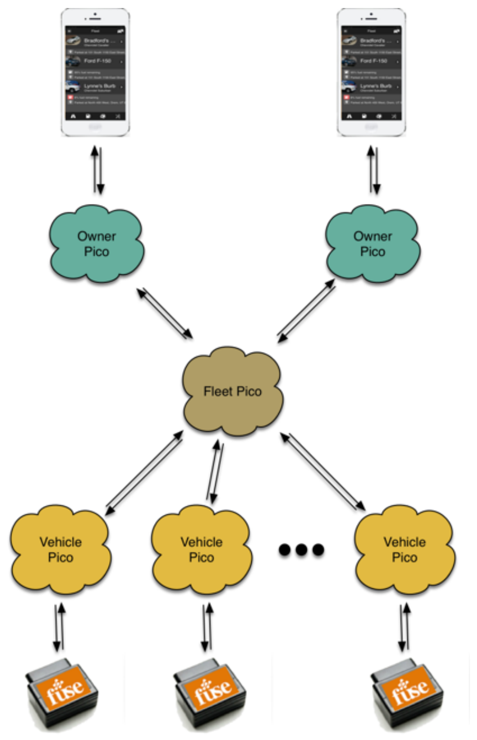
Fuse uses three types of picos:

- Owner picos represent the fleet owner and contain code that provides an API for interacting with an app as well as code for managing the fleet pico.
- Fleet picos represent the fleet and know how to interact with the owner pico as well as manage any number of vehicles. The state includes an eventually consistent representation of the state of the entire fleet.
- Vehicle picos represent each vehicle in the fleet. They listen for events from the physical OBD-II devices in the vehicle and interact with the fleet pico.

People using the app perceive the system as a monolithic service even though it comprises a number of independent parts acting in concert.



Fuse with Two Owners



Because Fuse is built using picos, the system is very flexible and can accommodate different usage patterns.

For example, it's common for more than one person to want manage a fleet (e.g. spouses). With a pico-based implementation, accommodating more than one owner is as easy as adding another owner pico to the configuration and ensuring it has the right attributes.

When either owner uses the app to connect to their pico, they are doing it with their own permissions, profile, and interaction state. There are no shared passwords.

When the fleet pico informs the owner of some important change in the state of the fleet it does so by attribute (e.g. *owner*) not name or channel and so all the owners get the same updates.

The behavior of the application depends on the relationships represented by the arrangement of channels, channel attributes, and the installed rules. Because there's no fixed schema or configuration, the overall application is very dynamic and flexible.

Events and Event Expressions

Picos use events internally and to communicate with one another

Events Make the Phone Ring

Request-based APIs are useful if you call them. But they aren't usually designed to notify clients of important events.

Event-based APIs, on the other hand are designed for inbound communication. An event-based API can send a message to interested parties when an important state change happens.

Events have important semantic differences from requests.

Event and Request Semantics

Requests and events differ in important ways. A request is made by a client asking the server to perform some action. An event, on the other hand, merely informs the recipient that something happened.

	Request-Based	Event-Based
Signal	Request receipt	Event receipt
Nature of signal	Please do this	Something happened
Obligation	At server	At sender
Interpretation	On client	On recipient

Events in Your Living Room

To see this, imagine you want your DVD pause the movie when you get a phone call. The following scenarios show the differences between implementing such a system using requests and events.

Request-Response System

The phone has an app that you configure to know about your DVD player so that whenever a phone call comes in, the app can use the DVD player's API to request that it pause.

In this scenario, the app must know that the DVD player exists and that it is capable of pausing. To work with multiple DVD players, there must be a complicated, standardized DVD API.

If we add more devices to this scenario like a stereo amp that should be muted, the app quickly becomes unwieldy.

In this scenario, the *semantic responsibility* for what should happen is concentrated in the mobile app. The app has to be very smart.

Event-Based System

The phone has an app that raises the `inbound_call` event whenever the owner answers a call. The DVD player is listening for events and pauses when it sees the event.

In this scenario, the app need only know that there is an event bus. Standardization is simply on common names for certain state changes (e.g. `inbound_call`).

Adding more devices to this scenario doesn't change the app on the phone at all—it doesn't even have to know they've been added.

In this scenario, the *semantic responsibility* is spread out between the devices and *encapsulated* in the device itself. This is an important form of isolation.

Event Schema

Events in picos have a particular structure that allows picos to exchange events without having to coordinate the syntax of the event.

Every event has an event domain and name. The domain can be thought of as the namespace that the designer uses to bound the semantics of a group of events.

The name (sometimes called a type) signifies the overall semantics of the event. Event names signify a state change and consequently are usually past-tense verbs.

Each event can have attributes that contain information about a that event. For example, an event that signals the end of a trip in Fuse might have attributes that specify the starting and ending location of the trip, the length, and the duration.

Event Expressions

An event expression, or eventex, is a declaration of a meaningful event scenario.

An eventex is introduced with the `select when` statement.

Simple event expressions have an event domain (e.g. `fuse`) and and event name (e.g. `vehicleCreated`).

More complicated eventexes can also test event attributes (e.g. `namespace, level`) using a regular expression or any boolean expression.

```
select when fuse vehicleCreated

select when cloudos subRequestPending
             namespace re/fuse-meta/gi

select when fuse fuelLevelUpdated
       where event:attr("level") > 20
```

Event Scenarios

An event expression can contain temporal relations to specify complicated event scenarios.

The first eventex in the example selects when the fuel is updated before the ignition status changes to "on".

The second looks for the last 5 `fuelLevelUpdate` events and averages the `level`, storing the running average in a variable named `avg_level`.

Note the use of regular expression value capture in the last example.

```
select when fuse fuelLevelUpdated
     before fuse ignitionStatus
             where status eq "on"

select when repeat 5
  (fuse fuelLevelUpdated level re/(.*)/)
  avg(avg_level)
```

Event Processing

Pico-based computation happens when rules process events and modify the internal state of the pico

Picos Take Action Through Rules

Picos are programmed by installing rulesets.

A ruleset contains declarations and rules.

The rules are called *event-condition-action* (ECA) rules because they take the form shown on the right.

Rules are selected when a particular event scenario occurs. If the condition is true, the rule fires and takes the action.

```
when  an event occurs

if    some condition is true

then  take some action
```

Picos Rule Structure

Rule syntax follows the ECA pattern.

The *event expression* is introduced in a *select* statement.

The *conditional action* follows a *prelude* where declarations are made.

The *postlude* can change pico state depending on whether or not the rule fired.

```
rule <name> {
  select when <eventex>
  pre {
    <declarations>
  }
  if <expr> then
    <action>
  fired {
    <effects>
  } else {
    <effects>
  }
}
```

Raising Events

Picos have an internal event bus. Any event sent to the pico or raised by an installed rule is potentially seen by every rule in the pico.

Any number of rulesets can be installed in a pico and a ruleset can contain many rules. The pico maintains an internal data structure called a *salience graph* that it uses to determine which rules are interested in which events.

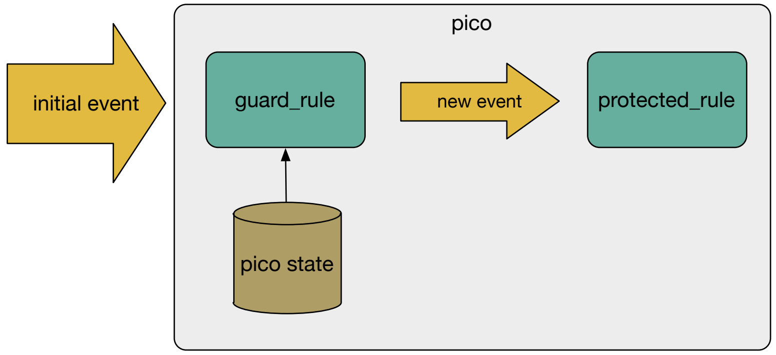
In addition to events sent to the pico via its channels, rules in the pico can raise events on the internal event bus.

Picos raise events in the rule's postlude. Raised events specify the event domain and name as well as any event attributes.

The condition in the rule determines whether the rule fires once it is selected. The postlude can be conditionally run based on whether or not the rule fires.

Rules raise events that are processed by other rules in the pico. This is a powerful programming technique. The following pages show some of the common patterns that occur in rule-based programming.

Guard Rules



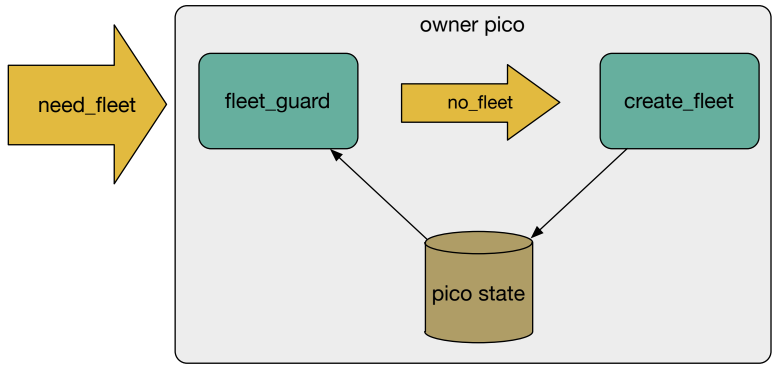
Guard rules are one of the most basic patterns in programming picos.

A guard rule selects on an incoming event and determines whether to raise an event that other protected rules in the pico will see.

Guard rules are useful in many situations:

- **Idempotency** – when a protected rule takes action that should only happen once, the guard rule can ensure it doesn't fire twice.
- **Initialization** – some rules must be run in a certain state. The guard rule can ensure the pico is initialized.
- **Authorization** – authorization (e.g. OAuth) is a special case of initialization.

Idempotence

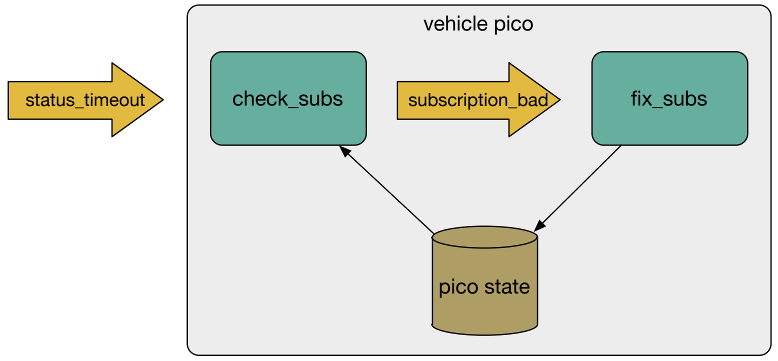


Idempotence is a particularly important property in reactive programming because the actor raising the event isn't privy to the internal state of the pico and so may raise an event multiple times.

For example, in Fuse (see introduction earlier in this lesson), the owner pico creates a fleet. We don't want multiple fleets, so a guard rule checks to see if a fleet already exists before raising the `no_fleet` event.

Because of the guard rule, the `need_fleet` event can be raised over and over again without adverse consequences.

System Checks and Self Healing



A variation on guard rules are rules that check certain properties of the pico and raise events that set off self healing procedures.

For example, in Fuse, the vehicle pico needs event channels, called subscriptions, for the physical device to raise events. If those become corrupted, the pico fails to get important events about the vehicle's state.

A `check_subs` rule runs periodically and if the subscriptions aren't right, it raises the `subscription_bad` event.

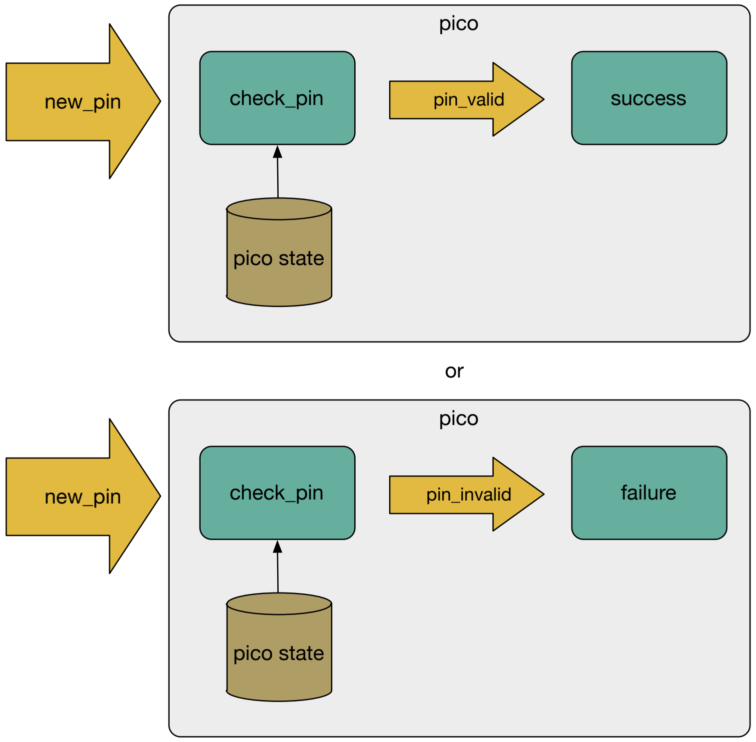
Event Splitting

We can use a rule to split an incoming event into multiple streams.

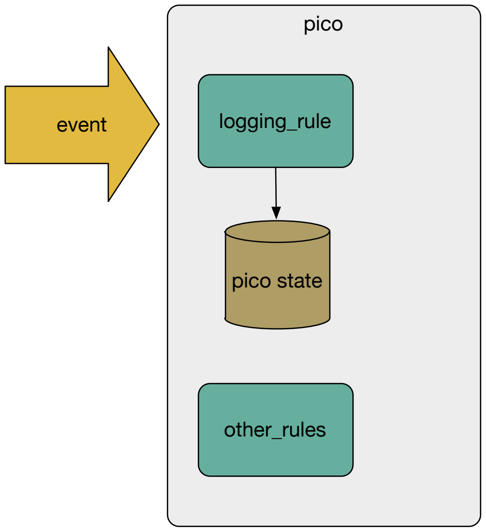
The event attributes, pico state, and even external data from APIs can be used to split a single event into several events.

For example, imagine that an event signals a login attempt and includes a pin as an attribute. The rule can check whether the pin is good or bad and raise events that correspond to these outcomes.

In the case of the pin, the outcomes are static, but a rule can calculate the event name to raise as well for dynamic routing.



Event Logging

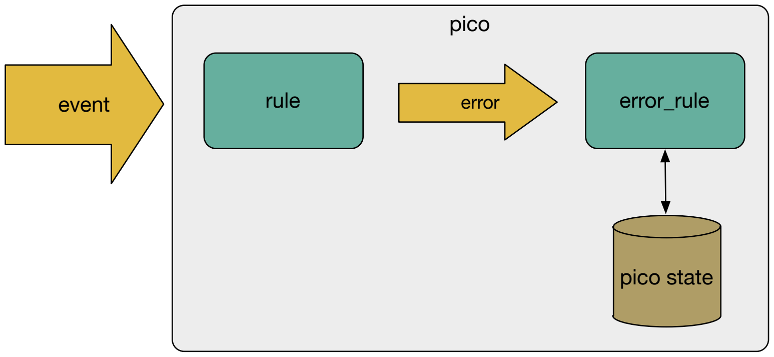


An event logger is a rule that logs an event by writing information about the event to the pico state.

Other rules in the pico can respond to the event in parallel. There's no reason to put an event logger in series with other rules.

An event logging rule might log all of the events of specific types, or just certain events based on their attributes. Logging can be useful for debugging and event replay.

Error Events



Certain errors cause error events to be raised automatically. In addition, a rule can explicitly signal an error and raise an error event.

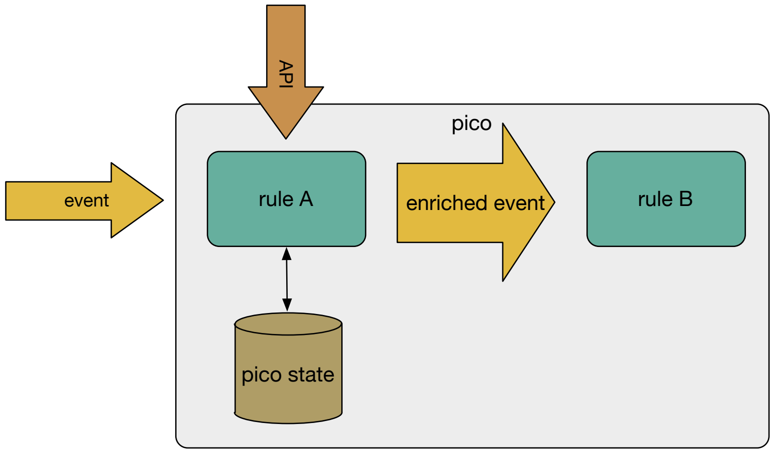
Rules can respond to error events. Such a rule might take action to correct the error, it might log the error, or it might email the administrator with the problem.

Error handling rules are first-class rules that respond to error events and thus are as capable as any other rule.

Enriching Events

Rules can enrich events, adding new information as attributes or changing attributes to reformat them.

A rule could, for example, get an event that a profile has been updated that doesn't include the changes, use an API to request the new profile, and then raise a new event with the updated profile information for other rules to use.



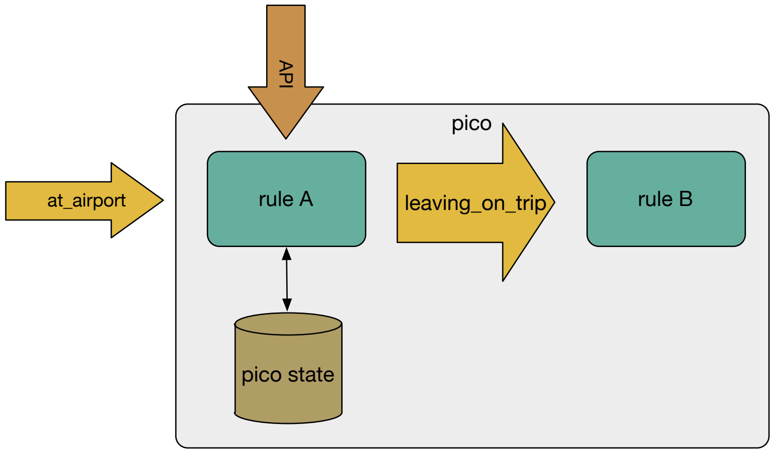
Semantic Translation

Semantic translation is a special kind of event enrichment.

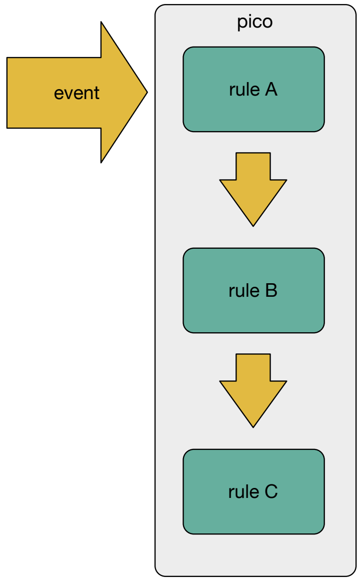
Semantic translation takes an event that means one thing, and turns it into an event that means something else based on information it knows about.

For example, an event that says you're at the airport could be turned into an event that says you're leaving on a trip, if the rule can confirm you have a ticket or your calendar shows a trip.

An event that signals leaving on a trip is more specific and thus more meaningful than one that merely indicates presence at the airport.



Event Chains and Pipelines



The preceding techniques can often be used together, creating a pipeline of rules that process events.

For example a guard rule, splitter rule, and enriching rule might be used in conjunction with each other.

I distinguish between pipelines and chains by the level of coupling that they exhibit. Chains are usually fairly tightly coupled and the rules in the chain are designed to work with each other and changing one often necessitates changing others. Guard rules are an example of event chaining.

Pipelines, on the other hand are more general purpose and are designed to work independently. The degree of coupling generally depends on the nature (including names) of events being raised. Narrowly named events combined with rules that select on that event are evidence of tightly coupled rules.

Chaining can be effective for specific situations, but recognize that the rules in the chain constitute a single computational unit because of their coupling.

Reactive Programming Patterns

Pico-based computation exhibits certain patterns that are useful in building reactive systems

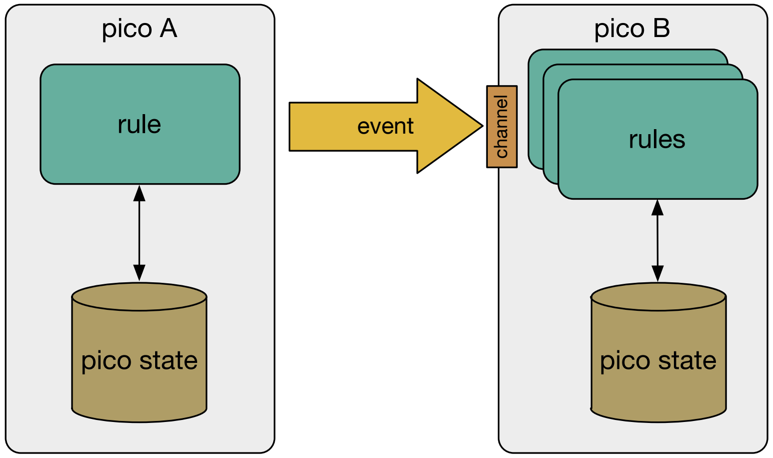
Pico-to-Pico Events

Pico-to-pico events are the primary way that picos interact.

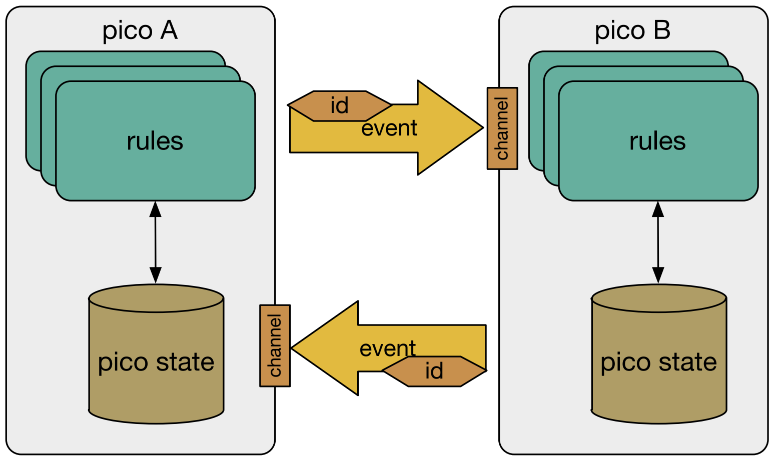
Picos *send* events using a rule action. Events are sent to a specific pico on a specific channel. Be sure to note the difference between *raising* an event on the internal event bus and *sending* an event to another pico.

Once the event is received by the pico, it is raised on the pico's internal event bus and any installed rule can select if its eventex is met.

As we've seen, the state of the two picos is completely isolated.



Correlation Identifiers



When one pico sends an event to another, it often is expecting an asynchronous response. A correlation identifier can be used to link these two events.

As we saw in **Lesson 7: Failure and Consensus**, a correlation identifier links conversational state in asynchronous interactions.

Correlation identifiers are passed as event attributes and can be any string that is unique within the conversations.

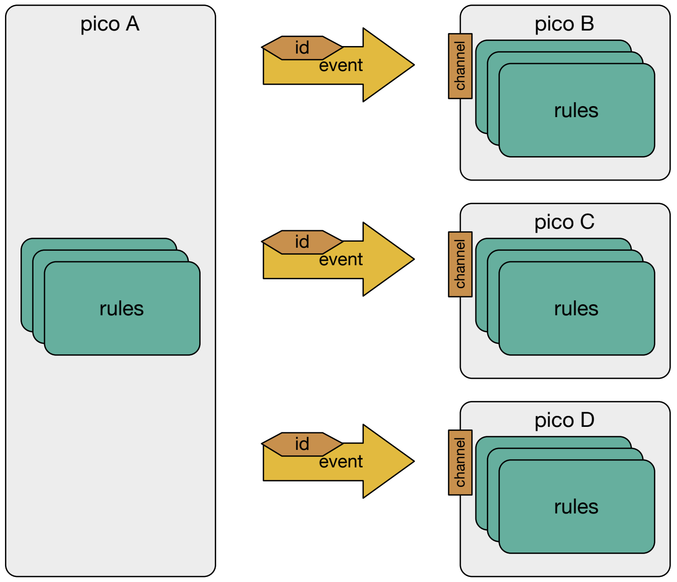
Rules use the correlation identifier to ensure that two processes don't get confused with one another. For example, the correlation identifier can be used in the pico's persistent state to keep data about different conversations separate.

Event Recipient Lists

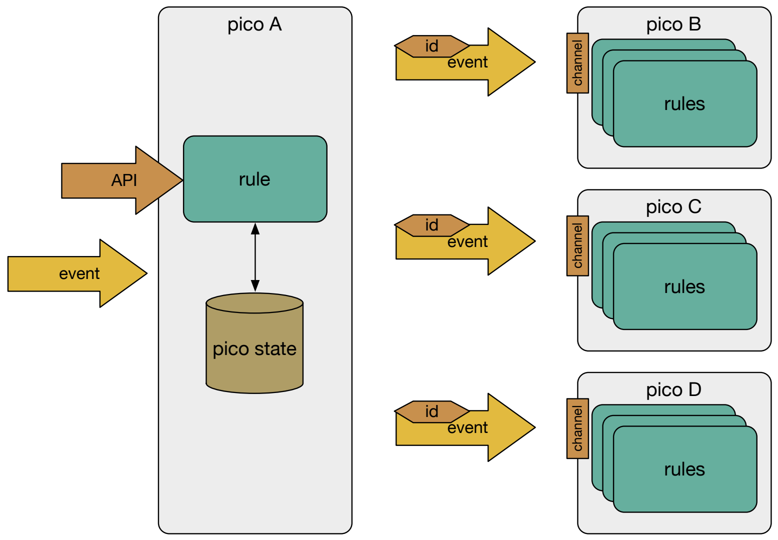
Picos can send events to each of the picos on a recipient list.

In this pattern one rule sends the same event to each of a number of other picos. The recipient list is analogous to the To: list on an email message.

The recipient list can be static, based on a particular configuration of picos in the computation, or it can be calculated in various ways. Computed recipient lists allow a pico to act as an *event router*.



Content-Based Event Routing



One way to compute the recipient list is to use the content of an incoming event.

The routing pico knows about some number of other picos and selects where to route the event based on the event domain, name, or attributes and other information such as the current state of the pico and external information from APIs.

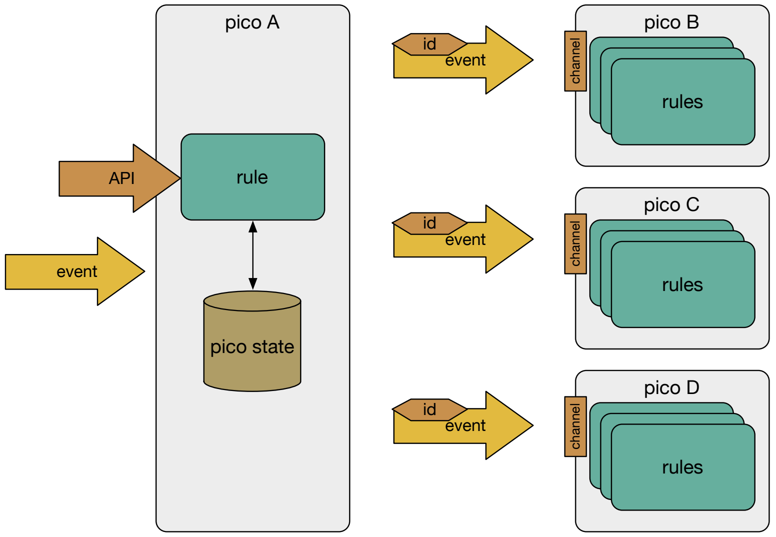
The routing rule usually attaches a correlation identifier to the event before routing it.

PubSub Event Routing

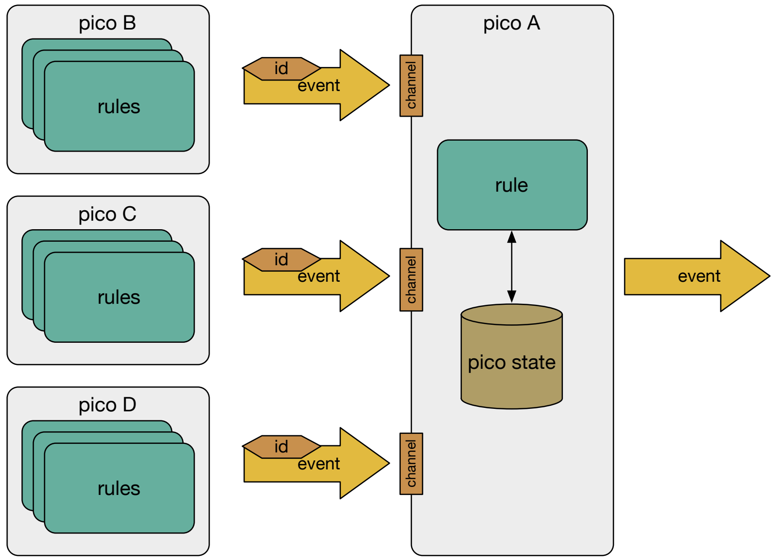
A variation on the content-based router is publish and subscribe, or *pubsub*.

In pubsub, the recipient picos are chosen because they have previously expressed interest in certain events.

The list of recipients is computed based on this interest. Recipient picos can unsubscribe from the events as well.



Aggregator

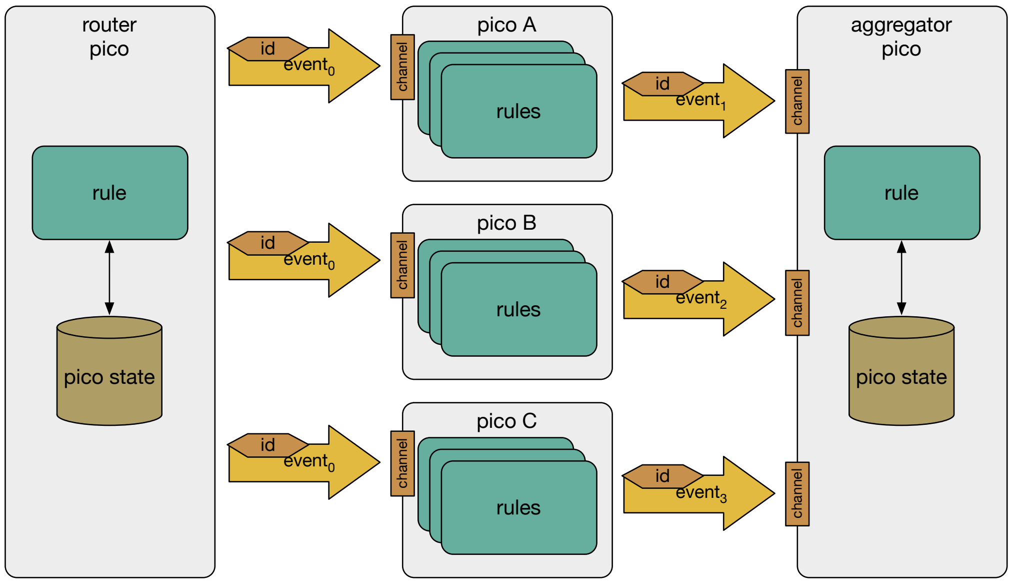


An aggregator is the dual of the event router.

In the aggregator pattern, a pico accepts events from some number of other picos and aggregates the results. Often the result will be that it sends an event to some other pico after it has seen the requisite incoming events.

The events usually have correlation identifiers to allow them to be aggregated with the correct conversational state.

Scatter-Gather



Combining the routing and aggregator patterns yields the scatter-gather pattern.

In scatter-gather, a pico sends an event to some number of recipients. Each recipient processes the event asynchronously and, when finished, sends an event to a pico that aggregates them.

The router and aggregator could be the same pico, but needn't be. The correlation identifier ensures that the picos can keep multiple, simultaneous conversations straight.

Handling Failure

Events sent from one pico to another can be lost for any number of reasons.

Event delivery is not guaranteed. Consequently, picos must use the same techniques we discussed in **Lesson 07: Failure and Consensus** to handle lost messages.

Sometimes it's OK if an event is lost. In Fuse, for example, sometimes end-of-trip events are lost. The system is designed to self heal and recover missing trips later.

Other times, the system must continue even if some picos failed to send the required event. In this case, a scheduled event can be used as a timeout to force the system to move on without the missing event.

Lastly, instead of merely moving on, you can design picos to use a scheduled event to cause a rule to resend the initiating event to picos that haven't yet responded. This is a good example of why idempotence is important in reactive programming.

Event Channels and Pico Security

Pico-to-pico communicate happens over *event channels*. Picos can have many event channels.

An event channel is point-to-point and delivers events directly to the pico. Picos can only interact by raising events and making requests on a specific event channel.

Picos can only get another picos' event channel in one of four ways:

1. Parenthood – Creating another pico
2. Childhood – Being created by another pico
3. Endowment – Receiving the channel at initialization
4. Introduction – Being introduced to another pico

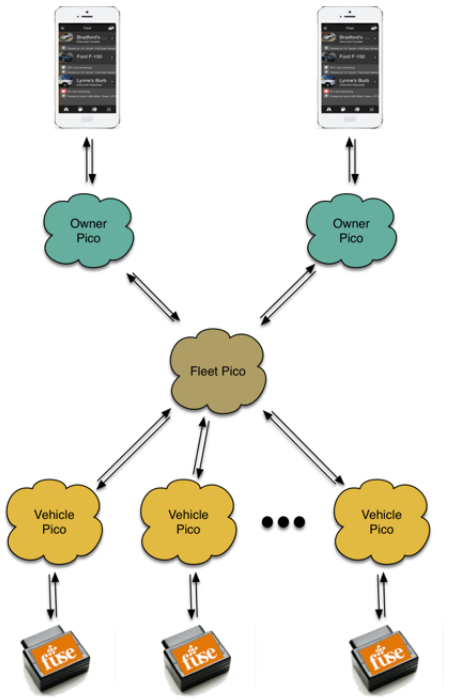
Children only have a channel to their parent, unless (a) the parent gives them other channels during initialization, (b) the parent introduces them to another pico, or (c) they are capable of creating children. Consequently, a pico is completely isolated from any interaction that its parent (supervisor) doesn't provide. This creates a security model similar to the object capability model.

Modeling with Picos

Picos are independent actors, representing independent entities.

Most reactive systems built from picos will comprise picos that closely correspond to entities in the design space.

The structure between the picos will correspond to the relationships that entities have in the modeled system. In Fuse, as we've seen, there are picos that represent the owner (a person), the fleet (a concept and collection), and each vehicle (a physical device).

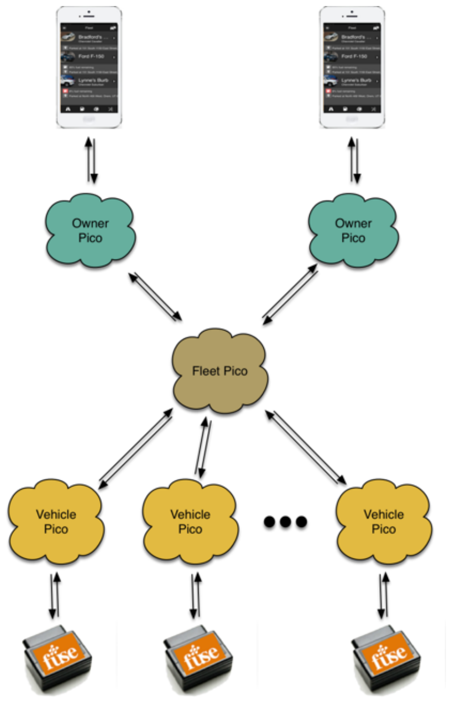


Thinking in Interactions

Computation in a pico-based system is the result of the interactions between loosely-coupled actors.

Going along with the idea of modeling with picos, developers have to think in terms of the interactions that picos have with each other. While the rulesets installed in each pico defines its behavior, the behavior of the application derives from the interaction that the picos have with each other.

Developers can use tools like pico maps that show those relationships and swim-lane diagrams that show the interactions that happen in response to events.



Conclusion

Summary & Review

Credits

Summary and Review

In this lesson we've seen that reactive programming, particularly using the Actor model, can be an antidote to the current Internet of Things architecture.

While the current Web 2.0-style of building the Internet of Things is interesting, it has inherent limitations.

Actors are independent programmed agents that can be arranged to model decentralized, heterarchical systems and interact asynchronously.

We explored a specific actor-model-based programming system called persistent compute objects, or picos. Picos can be used to model the relationships and interactions between independent entities.

We also discussed common patterns of interaction and saw how they allow us to build systems that are asynchronous, decentralized, heterarchical, and interoperable.

Credits

Photos and Diagrams:

All other photos and diagrams are either commonly available logos or property of the author.

- Goldfish (https://pixabay.com/en/fish-goldfish-underwater-water-sea-881161/), Public domain
- Pay phone (https://www.flickr.com/photos/67755197@N00/1096251053), CC BY-SA 2.0