Module 2: Motivation for Lingua Franca

Technical University of Vienna
Vienna, Austria, May 2022
References

Class website:

https://ptolemy.berkeley.edu/~eal/cps/

Lingua Franca website:

https://lf-lang.org/
The major challenge: Integrating complex subsystems with adequate **reliability**, **repeatability**, and **testability**.
Popular Techniques

- Publish and Subscribe
  - ROS, MQTT, Azure, Google Cloud

- Actors
  - Akka, Erlang, Orleans, Rebeca, Scala ...

- Service-oriented architecture
  - gRPC, Bond, Thrift, ...

- Shared memory
  - Linda, pSpaces, ...
Pub-Sub

- Components **publish** events on **topics**.
- Other components **subscribe** to topics.
- Message **handlers** are invoked in subscribers.
- No ordering guarantees.

ROS 2 (Robotic Operating System) uses pub-sub built on top of DDS (Data Distribution Service).
A ROS-Based Autonomous Driving Application: Autoware.Auto

ROS component architecture:

LGSVL simulation of the vehicle:

Soroush Bateni, of UT Dallas, studied this open-source system, which has been deployed on full-size cars.
Out-of-Order Message Handling

- **LGSL Interface:**
  1. Produce a “forward” gear
  2. Produce a (+) kinematic state
  3. Produce a “reverse” gear
  4. Produce a (-) kinematic state

- **Behavior Planner: What will it see?**
  - 2 -> 1 -> 3 -> 4
  - 1 -> 2 -> 4 -> 3
  - 1 -> 2 -> 3 -> 4
  - 2 -> 1 -> 4 -> 3

  Soroush ran 300,000 tests under benign conditions and found occurrences of all four sequences.

  The odd occurrences were rare enough that they are likely to not show up in testing!

Thanks to Soroush Bateni.
Port of Autoware.auto to Lingua Franca

Soroush Bateni
Popular Techniques

- Publish and Subscribe
  - ROS, MQTT, Azure, Google Cloud
- Actors
  - Akka, Erlang, Orleans, Rebeca, Scala ...
- Service-oriented architecture
  - gRPC, Bond, Thrift, Adaptive AUTOSAR, ...
- Shared memory
  - Linda, pSpaces, ...
Service-Oriented Architectures

- A **service** is a procedure exposed by a component.
- Components can invoke services on remote components.
- Caller may wait for results (**synchronous**) or retrieve results later (**future**).
- Service invocations are mutually exclusive, but there are no ordering guarantees.
Client SWC

Client Logic

method

invocation order on

services is not constrained.

Thanks to Christian Menard.

Service Proxy

SOME/IP

Service Skeleton

Server SWC

Server Logic

Client Code

```cpp
int main() {
    s = ServiceProxy();
    s.set_value(1);
    s.add(2);
    result = s.get_value();
    std::cout << result.get();
    return 0;
}
```
Emergency Brake Assist

Slide by Christian Menard.

[Diagram showing the process of Emergency Brake Assist (EBA) with stages including Video Provider, Video Adapter, Pre-processing, Computer Vision, and EBA.]
Popular Techniques

• Publish and Subscribe
  – ROS, MQTT, Azure, Google Cloud

• Actors
  – Akka, Erlang, Orleans, Rebeca, Scala ...

• Service-oriented architecture
  – gRPC, Bond, Thrift, ...

• Shared memory
  – Linda, pSpaces, ...
Actors are concurrent objects that communicate by sending each other messages.
Hewitt/Agha Actors

Data + Message Handlers

An actor with simple operations on its state:

Actor Foo {
    int state = 1;
    handler dbl()
    
        state *= 2;
    
    }
    handler inc(arg)
    
        state += arg;
    print state;
    
    }
    }
Example

An actor that uses actor Foo:

```java
Actor Bar {
    handler main() {
        Foo x = new Foo();
        x.dbl();
        x.inc(1);
    }
}
```

Semantics is “send and forget.”
What is printed?

Actor Bar {
    handler main() {
        Foo x = new Foo();
        x.dbl();
        x.inc(1);
    }
}

Actor Foo {
    int state = 1;
    handler dbl() {
        state *= 2;
    }
    handler inc(arg) {
        state += arg;
        print state;
    }
}
Pass-Through Actor

Baz: Given an actor of type Foo, send it “double”:

```scala
Actor Baz {
  handler pass(Foo x) {
    x.dbl();
  }
}
```
Actor Bar {
    handler main() {
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.inc(1);
    }
}

Actor Baz {
    handler pass(Foo x) {
        x.dbl();
    }
}

Actor Foo {
    int state = 1;
    handler dbl() {
        state *= 2;
    }
    handler inc(arg) {
        state += arg;
        print state;
    }
}

What is printed?
What assumptions are needed for it to be safe for the open_door handler to open the door?

Actor Source {
    handler main() {
        x = new Door();
        x.disarm_door();
        x.open_door();
    }
}

Actor Door {
    handler open_door() {
        ...
    }
    handler disarm_door() {
        ...
    }
}
Now what assumptions are needed for it to be safe for the open_door handler to open the door?

Actor Source {
    handler main() {
        x = new Door();
        r = new Relay();
        r.check();
        x.open_door();
    }
}

Actor Relay {
    handler check(Door x) {
        x.disarm_door();
    }
}

Actor Door {
    handler open_door() {
        ...
    }
    handler disarm_door() {
        ...
    }
}
Hewitt/Agha Actors are Not Predictable

Messages are handled in nondeterministic order.
One Solution: Analyze and Use Dependencies

But how? Where is the dependence graph?

Actor Source {
    handler main() {
        x = new Door();
        r = new Relay();
        r.check();
        x.open_door();
    }
}

Actor Door {
    handler open_door() {
        ...
    }
    handler disarm_door() {
        ...
    }
}

Actor Relay {
    handler check(Door x) {
        x.disarm_door();
    }
}
One Solution: Analyze and Use Dependencies

And what if the dependence graph is data dependent?

Actor Source {
    handler main() {
        x = new Door();
        r = new Relay();
        r.check();
        x.open_door();
    }
}

Actor Door {
    handler open_door() {
        ...
    }
    handler disarm_door() {
        ...
    }
}

Actor Relay {
    handler check(Door x) {
        if (something) {
            x.disarm_door();
        }
    }
}

Return to simple, concrete example

How to achieve deterministic behavior?

Actor Bar {
    handler main() {
        Foo x = new Foo();
        Baz z = new Baz();
        z.pass(x);
        x.inc(1);
    }
}

Actor Foo {
    int state = 1;
    handler dbl() {
        state *= 2;
    }
    handler inc(arg) {
        state += arg;
        print state;
    }
}

Actor Baz {
    handler pass(Foo x) {
        x.dbl();
    }
}


Part 1 of our Solution: Ports

reactor Bar {
    output dbl:bool;
    output inc:int;
    reaction (startup) -> dbl, inc {=
        lf_set(dbl, true);
        lf_set(inc, 1);
    =}
}

Instead of referring to other actors, an actor refers only to its own ports (and ports of contained reactors).

reactor Baz {
    input in:bool;
    output out:bool;
    reaction (in) -> out {=
        lf_set(out, in);
    =}
}
Part 1 of our Solution: Ports

Input ports look like the message handlers of actors.

```cpp
reactor Foo {
  input dbl:bool;
  input inc:int;
  state s:int(1);
  reaction(dbl) {=
    self->s *= 2;
    =}
  reaction(inc) {=
    self->s += inc.value;
    =}
}
```
main reactor {
    b = new Bar();
    r = new Baz();
    f = new Foo();
    b.dbl -> r.in;
    r.out -> f.dbl;
    b.inc -> f.inc;
}
Part 3 of our Solution: Scheduling

main reactor Top {
  x = new Foo();
  y = new Bar();
  z = new Baz();
  y.double -> z.in;
  y.increment -> x.increment;
  z.out -> x.double;
}

Scheduling becomes especially interesting when production or consumption of messages is data dependent.

Ensure that Baz completes before Foo’s handlers are invoked.
Active, Ongoing Project

https://repo.lf-lang.org

Plenty of “good first issues”

Give us some ⭐️’s

Contribute
Slack Workspace

I have created a Slack workspace called:

lf-community

You should have gotten an invitation to join it. Please use it for discussions, questions, and problems with Lingua Franca.
• Pub-Sub, SoA, and Actors are all problematic.
• The problems are solvable (Lingua Franca).