Data-centric Invocable Services

Workshop on Real-Time, Embedded and Enterprise-Scale Time-Critical Systems
April 2012; Paris

Rick Warren
rick.warren@rti.com
Rajive Joshi
rajive@rti.com
Data-Centricity by Example: Calendaring

Imperative Process:
1. Email: “Meet Monday at 10:00.”
2. Email: “Meeting moved to Tuesday.”
3. Email: “Here’s conference call info…”
4. You: “Where do I have to be? When?”
5. You: (sifting through email...)
Data-Centricity by Example: Calendaring

Data-centric Process:
1. Calendar: (add meeting Monday at 10:00)
2. Calendar: (move meeting to Tuesday)
3. Calendar: (add dial-in info)
4. You: “Where do I have to be? When?”
5. You: (check calendar)
Data-centric vs. Imperative Styles

• **Imperative**
  – Tell someone to do something
  – Check whether they did it

• **Data-centric**

<table>
<thead>
<tr>
<th>Canonical</th>
<th>SQL</th>
<th>HTTP</th>
<th>DDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>INSERT</td>
<td>POST</td>
<td>write</td>
</tr>
<tr>
<td>Read</td>
<td>SELECT</td>
<td>GET</td>
<td>read</td>
</tr>
<tr>
<td>Update</td>
<td>UPDATE</td>
<td>PUT</td>
<td>write</td>
</tr>
<tr>
<td>Delete</td>
<td>DELETE</td>
<td>DELETE</td>
<td>dispose</td>
</tr>
<tr>
<td>on_event</td>
<td>TRIGGER</td>
<td>—</td>
<td>on_data_available</td>
</tr>
</tbody>
</table>

Functions coupled; State implicit

Functions decoupled; State explicit
Conflict?

• **Data-centricity benefits system designers**
  – (Eventually) Consistent “truth”
  – Scalability and elasticity, because of stateless logic
  – Evolvability, because of functional decoupling
  – Robustness to communication failures

• **Programmers think imperatively**

```java
try {
    doSomethingThatMayFail();
    doSomethingOnSuccess();
} catch (FailureOccurred) {
    doSomethingElseOnFailure();
}
```
Conflict?

Are these models fundamentally in conflict?

No.
Reconciliation

“Dude, instructions are just data.”
— John Von Neumann, 1945

- *Desires* are also just data
- *Objectives* are also just data

**Best practice:** Don’t tell someone to do something; assert your desired future state
Reconciliation

Imperative View

```java
try {
    doSomething();
    doOnSuccess();
} catch (Failure) {
    doOnFailure();
}
```

Data-centric View

```java
struct ThingDesired {
    string thingWanted;
}

struct ThingHappened {
    string thingWanted;
    boolean success;
}

...
Reconciliation

Imperative View

```java
try {
    doSomething();
    doOnSuccess();
} catch (Failure) {
    doOnFailure();
}
```

Data-centric View

```java
...
create(
    thingWanted = "something")

on_event(
    thingWanted == "something"
    &&
    success == true) {
    update(
        thingWanted = "onSuccess")
}

on_event(
    thingWanted == "something"
    &&
    success == false) {
    update(
        thingWanted = "onFailure")
}
```
Reconciliation

Imperative View

This way is easier
...to write the first time
...to understand top-down, sequentially

Data-centric View

This way is easier
...to scale out
...to extend
...to maintain incrementally
...to understand bottom-up, causally
Imperative View

Express application code this way

Data-centric View

Express interop. this way

This is what we mean by “data-centric invocable service”:
- Request looks like explicit “call” (RMI or similar)
- Interoperability on the basis of data, not functional implementation
- *Caller and Callee may use different programming models*
Reconciliation

<table>
<thead>
<tr>
<th>Imperative View</th>
<th>Data-centric View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Express application code this way</td>
<td>GOOD</td>
</tr>
<tr>
<td>Make app programmers wire that up</td>
<td>HARD</td>
</tr>
<tr>
<td>Express application code this way</td>
<td>BAD</td>
</tr>
</tbody>
</table>

Express interop. this way

Express interop. based on one app’s code
Process

1. Define data model
2. Define and group data streams using those types
3. Define deployment of producers and consumers of those streams

4. Generate code from architect-defined contract(s) (OPTIONAL)
5. Produce and respond to data according to app algo’s

6. Govern data flows based on architect-defined contract(s)
Process Example: WSDL

class MyService {
   Result1 doThis(Args1) {
      ...
   }

   Result2 doThat(Args2) {
      ...
   }
}

Standard WSDL bindings exist for SOAP, REST, and SMTP.

Additional bindings could be defined for DDS or other technologies.

Figure courtesy of ICONIX,
http://www.iconixsw.com/Articles/SOARoadmap.html

© 2012 RTI
Hypothetical DDS IDL Example

```idl
struct Foo {
    @Key long id;
    long x;
    double y;
};

struct Bar {
    @Key string who;
    short baz;
};

dclocal interface Svc {
    @InTopic("FooTpc")
    @OutTopic("BarTpc")
    Bar doFoo(in Foo f);
};
```

Control binding to topics for easy subscription, filtering, and QoS control
Hypothetical DDS IDL Example

```idl
struct Foo {
    @Key long id;
    long x;
    double y;
};

struct Bar {
    @Key string who;
    short baz;
};

local interface Svc {
    @InTopic("value="T1", key="42")
    @OutTopic("value="T2", key="me")
    Bar doFoo1(in Foo f);

    @InTopic("value="T1", key="29")
    @OutTopic("value="T2", key="you")
    Bar doFoo2(in Foo f);
};
```
local interface Svc {
    Foo doFoo(
        out string x,
        inout long y,
        in double z);
    Bar doBar(
        in long a,
        out float b,
        inout short c);
};

union Svc_Type
switch (WhichOpEnum) {
    case Svc_doFoo_CALL:
        Svc_doFoo_InStruct sdfis;
    case Svc_doFoo_RETURN:
        Svc_doFoo_OutStruct sdfos;
    case Svc_doBar_CALL:
        Svc_doBar_InStruct sdbis;
    case Svc_doBar_RETURN:
        Svc_doBar_OutStruct sdbos;
};
Why is This Gross?

- Insane data type to make a DBA cringe in horror
- Segregates data model: human-written vs. machine-generated
- (Someone’s) API constraints baked into shared interop. interface
- Heavily influenced by scalability assumptions...which may or may not be valid
- Over-reliance on complex filtering on subscriber side
- Fine-grained QoS governance almost impossible

```c
union Svc_Type
switch (WhichOpEnum) {
  case Svc_doFoo_CALL:
    Svc_doFoo_InStruct sdfis;
  case Svc_doFoo_RETURN:
    Svc_doFoo_OutStruct sdfos;
  case Svc_doBar_CALL:
    Svc_doBar_InStruct sdbis;
  case Svc_doBar_RETURN:
    Svc_doBar_OutStruct sdbos;
};
```
Summary

• Data-centricity remains an architectural best practice
  – Many of its benefits accrue at the systems level

• App programmers may be more comfortable with higher-level API

• These two sets of benefits are not exclusive
  – Design system interop. interface
  – Design programming interface
  – Bind them together

• Proven examples exist:
  – Architecture
  – Workflow
  – Data and service description
  – Programming model