Motivation – our use case

- Information interoperability enables access to embedded information and mashups
- Two use cases at the same time: Music Follows User, Read aloud message
Service and semantic level

- Service level: interfaces to service implementations
- Semantic level: access to semistructured, self-describing information (RDF)

Semantic level problem

- Mash ups easy when just writers and one reader
- Truism: two way communication requires an agreement or protocol
- Standardization slow and heavyweight -> at information level standardize on RDF
Semantic level problem for use case

• Who gets to use the **limited resource** of noisemaking?
**Straightforward Attempt**

• Simply subscribe to the message

![Diagram](image)

```
0x123 a Msg
:content "Foo"
:isread "no"
:target +35850
```

---

**Straightforward Attempt**

• Update and synchronize on a property, shared between player and reader

• What if player should be paused by something else than Msg
  • modify player to understand the new thing?
  • abuse Msg as mechanism to stop the player?

• -> needs Separation of concerns
Abstraction

Concrete/Service

0x987 a AudioCap
:commits 0x456

0x123 a Msg
:content "Foo"
:target +35850

Abstract/Semantic

Player

0x456 a Activity
:uses 0x987
:importance "music"
:active "yes"
:requires ":AudioCap"

Reader

0x789 a Activity
:uses 0x987
:importance "traffic"
:active "no"
:requires ":AudioCap"

Abstraction For the "user"

• User creates an instance and promises to observe and honor ":active"
• It targets resources by :uses properties
• It may require types of resources
• An Activity instance is "virtual"
Service view

• A service implementation must represent itself semantically
• It may be pointed by an activity with :uses, to which it can reply with a :commit
• The service implementation may accept or deny :uses **locally**

```
0x987 a AudioCap
  :commits 0x456
```

```
0x456 a Activity
  :uses 0x987
  :importance "music"
  :active "no"
  :requires ":AudioCap"
```

---

Service view

• A service implementation can interpret the additional information provided by Activity to its benefit or detriment
• It may or may not publish its criteria
• Or use available defaults

```
max_users 1
traffic > music
```

```
0x987 a AudioCap
  :commits 0x456
```

```
0x456 a Activity
  :uses 0x987
  :importance "music"
  :active "yes"
  :requires ":AudioCap"
```

```
0x789 a Activity
  :uses 0x987
  :importance "traffic"
  :active "no"
  :requires ":AudioCap"
```
Local and co-operative scheduling

• An activity is responsible for setting itself active and inactive
• Active “yes” when all :uses have :commits

max_users 1
traffic > music

Local and co-operative scheduling

• Local decisions
  • Service implementations automatically yield to higher priorities
  • Activities locally monitor their status
• Are not always enough:

• Timeout and retry for Activities is a localized solution
• A global arbiter can also be implemented, but may not be scalable
• May not be fair…
The End  -- Thanks

• We hope:
  • reasonable compromise between standardization and ad-hoc approach
  • lightweight and simple enough for Node and application implementations
  • saves our own work in implementing new use cases and scenarios

• Answer set programming rules implementing described arbitration available at http://sourceforge.net/projects/ssl

• We expect:
  • identify (and implement) other similar patterns
  • contribute to core ontologies along with behaviour implementations

• Question:
  • Here deliberate separation of Activity from Content (e.g. SMS)
  • RDF enables same instance to be interpreted as different Classes
  • Would it make sense always combine the Content and the Activity as same instance?