Tuples on the Ambient (TOTAM)

Tuple space model targeting mobile devices combining the best of two worlds:

federated tuple space model

- guarantees atomicity for *in* operation.
- reactions.

replication model

- *rd* operation fully decoupled in time.
- scoped propagation protocol.
- antituples.

Tuples

```
def myTupleSpace := makeTupleSpace();
// a “hallo” message tuple from wolf.
def halloTuple := tuple: [Message, “wolf”, “hallo”];
// a template for message tuples from wolf.
def wolfTuples := tuple: [Message, “wolf”, var: `content];
// a template for any message tuples.
def msgTuples := tuple: [Message, var: `from, var: `content];

// add tuple to tuple space
myTupleSpace.inject: halloTuple;
// get a Message tuple
def aMessageTuple := myTupleSpace.rdp(msgTuples);
// get all Message tuples.
def messageTuples := myTupleSpace.rdg(msgTuples);
```

Non-blocking operations

Reactions on Tuples

once and forever reactions by means of when: and whenever: listeners

Variables are bound

```
//read a Message tuple
myTupleSpace.whenever: msgTuples read:{
  system.println(“Got message: “ + content + “ from: ” + from);
};
//remove all Message tuples from wolf
myTupleSpace.whenever: wolfTuples in:{
  system.println(“Wolf says: “ + content);
};
```

Returns object to cancel the subscription

Tuples on the ambient

Network facilities disabled by default!

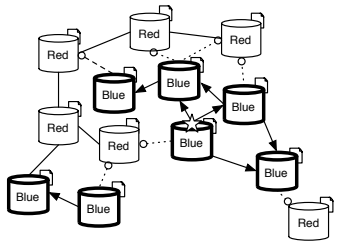
```
myTupleSpace.goOnline();
myTupleSpace.inject: halloTuple;

def defaultPropagationProtocol(){
  isolate: {
      //receiver-side protocol
     def decideEnter(ts) { true };
     def doAction(ts){};
     def changeTupleContent(ts){self};
     def decideStore(ts) {true};
      //sender-side protocol
     def inScope(senderDescriptor,receiverDescriptor){ true };
     def decideDie(ts){false};
   };
};
```

Carries the default propagation protocol

Scoped Tuples on the Ambient

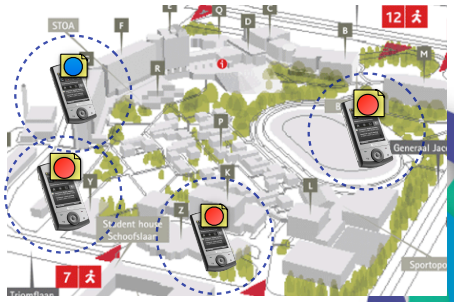
- Each tuple space has a tuple space **descriptor**.
- Tuples are propagated to neighbours **in the scope of** the tuple.



Flikken

Today's assignment:

1. Players can track the position of nearby team members on a map.
2. When a player does not update its position for a certain amount of time, its visual representation on the map should be grayed out.
3. Players die when they step into a mine virtual object.



Conditional Synchronization (CS)

- with Futures:

```
def testAsyncOneCustomer(){
  def future := when: customer<-haveComputer()@FutureMessage becomes:{
    |val|
    self.assertEquals(val, 1);
  };
  future;
};
```

- applying the `becomes:` block resolves `future`.
- applying the `catch:` block ruins `future`.

CS with Futures

- Synchronization based on event or conditions by explicit future manipulation:

```
def [future, resolver] := makeFuture();
consumer<-give(future);
def val := /* calculate useful value */
      resolver.resolve(val);
      resolver.ruin(exception);
```