Prototype Platforms for Distributed Agreements

Dipartimento di Informatica
Università di Pisa

Alberto Baragatti, Roberto Bruni, Hernán Melgratti,
Ugo Montanari and Giorgio Spagnolo
Motivation

- Orchestration of multi-party negotiations
  - Participants cannot be fixed statically
  - Participants have a partial view of the whole set of parties
- To develop a coordination pattern that exploits the D2PC protocol for orchestrating agreements
- To build a prototype application
Scenario: Rescue teams
Scenario: Rescue teams

Wants to assign the activity A_1 to Leader_1
Scenario: Rescue teams

Perform A_1
Scenario: Rescue teams

Decides to assign $A_1$ to Op_1 and Op_2
Scenario: Rescue teams

Perform A_1
Scenario: Rescue teams

[Diagram showing network connections between Central Base, Leader_1, Leader_2, Op_1, Op_2, and Op_3. Arrows indicate communication paths.]

- Leader_1 accepts the task.
- Leader_2 refuses the task.
Scenario: Rescue teams

Attempts to assign A_1 to Op_3
Scenario: Rescue teams
Scenario: Rescue teams

Accepts the request
Scenario: Rescue teams

Central Base

Leader_1

Op_3

Leader_2

Op_1

Op_2

Accepts the request
Scenario: Rescue teams

Exceptions:

- The Central realizes that A_1 is not longer needed
- The Leader_1 prefers to execute another activity
- Op_2 is unable to do A_1
Scenario: Rescue teams

Exceptions:

- The Central realizes that A_1 is not longer needed
- The Leader_1 prefers to execute another activity
- Op_2 is unable to do A_1

We need an agreement mechanism!
Agreements

- The structure of agreements depends on the interactions among the different parties.
- Participants can dynamically join a negotiation.
  - Operators and leaders are getting involved in a negotiation during the execution of the agreement.
  - Neither the number nor the identity of parties are known statically.
- Asynchronous communication.
Coordination Pattern

- We rely on the *Distributed Two Phase Commit* (D2PC) of [BLM2002]:
  - A variant of the decentralized 2PC.
  - Finite but unknown number of participants.
  - A participant P ready to commit has a partial view of the set of participants
    - Only those who directly cooperated with P
  - P contacts all known partners and learns the identity of other participants from them.
D2PC: Initial state
D2PC: Initial state
D2PC: Initial state
D2PC: Running
D2PC: Running
D2PC: Running
D2PC: Running
D2PC: Running
D2PC: Running
About the D2PC

- It has been specified in the Join Calculus.
- It was proposed to encode zero-safe nets in Join.
- It has been proved to be correct when there are no failures.
Coordination Pattern
1. **Initialization:**
   Any participant creates a coordinator to handle the agreement.
Coordination Pattern

2. Application Logic:
Participants interact and exchange the identities of their coordinators
Coordination Pattern

2. Application Logic:
Participants interact and exchange the identities of their coordinators
2. Application Logic:
Participants interact and exchange the identities of their coordinators
Coordination Pattern

3. Start of the D2PC:
Participants start the commit protocol either voting commit or abort
Coordination Pattern

4. Execution of D2PC: Coordinators eventually arrive to an agreement
Coordination Pattern

5. Communication of the result:
Coordinators notifies the application with the result of the agreement
A prototype implementation for a minimal set of functionalities:
- Users exchange textual messages.
- Users can decide either to commit or to abort.
- Users see the outcome decision.

Parties have been developed in:
- Jocaml + Perl running on Linux.
- Polyphonic C# (or Comega) running on .Net.
- They can interact, i.e., participate in a negotiation.
Implementation (2)

- Any party is identified with a **unique ID** (provided when the application is launched).

- A **configuration file** associates IDs to IP addresses.

- The **ports** in which parties communicate depend exclusively on the ID.
Implementation (3)

- Parties communicate through TCP sockets
Implementation (3)

- Parties communicate through TCP sockets
Implementation (3)

- Parties communicate through TCP sockets
Perl+Jocaml Components

- Three-Layer Architecture

![Diagram showing a three-layer architecture with Perl and Jocaml components. The layers are labeled GUI, Coordinator, and D2PC. Red arrows indicate the flow between Perl, Jocaml, and the layers.](image-url)
Perl+Jocaml Components

- Three-Layer Architecture
Perl+Jocaml Components

• Three-Layer Architecture
Perl+Jocaml Components

- Three-Layer Architecture
Perl+Jocaml Components

• Three-Layer Architecture
Jocaml Coordinators

- Jocaml is an extension of Ocaml with:
  - **Processes**: Expressions + Async Messages.
  - **Channels**, i.e., join ports
  - **Join patterns**

```ocaml
let def a! h | b! () =
    if h < 5 then c() else d();
```
let def d2pc () =
  let def
    state! h | abt! () = failed() | fwdabt [h]
  or failed!() | abt! () = failed()
  or failed!() | lock! (l1,l,a) = failed() | fwdabt [a]
  or failed!() | put!(l,a,c) = failed() | fwdabt a
  or commit!(l,l1,l2,c,a) | abt!() = failed() | fwdabt a
  or commit0!(l,l1,l2,c,a) =
    match l with
    [] -> if (equiv l1 l2) then fwdcmt [c]
    else commit(l,l1,l2,c,a)
    t::ts -> fwdlock(t,l1) | commit0(ts,l1,l2,c,a)
  or commit!(l,l1,l2,c,a) | lock!(l3,l1,f) =
    commit0 (difference l3 l1, union l1 l3,
        union l2 [l1],c,union a [f])
  or state! h | put! (l,a,c) =
    commit0 (del lock l, l, [lock], c,union a h)
  in reply lock,put,commit,state;;
Polyphonic C# Components

Polyphonic C# extends C# with:

- **Asynchronous methods**: any call is guaranteed to complete almost immediately.

- **Chords** defined by:
  
  - A *header*: a set of method declarations separated by `&`.
  
  - A *body*, which is executed only once all the methods in the header have been called.
Polyphonic C# Components (2)

- Polyphonic C# class for the D2PC

```csharp
public class D2PC{
    public async put (listhost l, port a, port c);
    public async abt();
    public async mlock(listhost ll, port l, port a);

    private async state(port h);
    private async failed();

    ....
    when state(port h) & abt(){failed () ; ... }
    when failed() & abt(){failed() ;}
    when failed() & mlock(listhost ll, port a, portl){
        failed();
    }
    ...
}
```
Polyphonic C# Components (3)

- Class Diagram (partial view).

```
Participant

Receiver
  static async listen(...)

GUI

Sender
  static async sendMsg(...)

D2PC
  async put(...)
  async abt(...)
  async mlock(...)```
Demo ...
Future work

- Extending the D2PC for handling failures.
- Combining the D2PC with the standard 2PC protocol.
- Adding a mechanism for the dynamic discovery of participants (instead of using a configuration file).
- Implementing all the functionalities of the rescue team scenario.