Requirements Engineering for Multi-Stakeholder Distributed Systems at Hooli

Presented by: Andrew Pham
Overview

1. What Is a Distributed System?
2. Requirements Engineering for Distributed Systems
3. Framework for Requirements Engineering of Distributed Systems
4. Modelling Techniques for Our RE Framework
5. Synthesizing Stakeholder Requirements
6. Synthesizing Towards an Open System
7. Monitoring for Change
8. Adapting the Service in Response to Change
9. A Real World Example
10. A Result from the Initial Design/RE Phase (Architecture Prescription)
11. Service Discovery and Adding Details and Granularity to the Design (Solution Architecture)
12. Keys to Successful Requirements Engineering
13. Postmortem
14. Helpful Tips
What Is a Distributed System?

- A network of individual web services, each of which is designed, owned, and operated by a distinct stakeholder;

- **Service-Oriented Architecture (SOA):** The key to high scalability across the company; and

- The key to growing software in a distributed fashion.
Requirements Engineering for Distributed Systems

- Requirements engineering (RE) when relying on services or dependencies that may or may not exist in the distributed system (or that you have no control over) is HARD!

- How can we mitigate the complexities of SOA and simplify the problem so that the direct engineering team can tackle the problem head on?

- *Three proven RE approaches* are (1) goal-oriented requirements modeling, (2) variability modeling, and (3) negotiation techniques.
Framework for Requirements Engineering of Distributed Systems

Fig. 1. Framework for MSDS RE (adapted from [8]).
Modelling Techniques for Our RE Framework

*i* Modelling Language

- This approach is extended and complemented with *variability modelling techniques* from product line engineering and *negotiation techniques* known from requirements engineering.

- Goal-oriented models like the i* language are a good choice because they allow for the expression of high-level requirements (Fig. 2). The i* notation is not only goal-oriented but actor-oriented as well.

- Actors are mainly used to represent *stakeholders*, while in lower levels of our framework, they stand for *services and components* (Fig. 5).
Modelling Techniques for Our RE Framework

**Negotiation Techniques**

- Win-win negotiation allows for the expression of stakeholder goals as win conditions.
- Resolve conflicting win conditions by proposing options and alternatives to overcome the identified problems.
- The goal is to cover all win conditions by agreements.
Modelling Techniques for Our RE Framework

**Variability Modelling**

- Based on the concept of variability in product line line development.
- A variation point represents an explicit engineering decision to allow for several alternative variants.
- Product line concepts such as variability are increasingly used to support run-time evolution and dynamism.

![Variation Point Diagram](image-url)
Synthesizing Stakeholder Requirements

- The main objective is to develop a requirements model capturing the needs of the various stakeholders.
- Identifying the right set of actors covering these stakeholders is key.
- Conflicting requirements are especially pernicious in distributed systems.
- **Solution:** Start with the information gathered on the stakeholder needs layer—i.e., the i* models expressing the goals of the different stakeholders. Then, identify their wishes and needs. Then, negotiate their resolutions.

![Diagram of stakeholder needs](image)
Synthesizing Towards an Open System

- Next, we map stakeholder goals onto architectural concepts on the architectural prescription layer of the framework (the different model elements of i* are applicable in the three lower levels).

- **Example:** At the *architecture prescription* layer, there may be two different actors (i* roles) for the types of services—“Flight Ticket” and “Hotel Booking” (Fig. 5).

- **Example:** In the *solution architecture*, we may choose to cover these roles with one actor—“Amadeus” (Fig. 6).

- **Example:** In the *open system* layer, we may choose the “Amadeus” service hosted on the Spanish site (Fig. 7).
Monitoring for Change

- Distributed systems tend to change a lot.
- Continuous monitoring is necessary to ensure that requirements of stakeholders remain satisfied after changes.
- In our framework, we need to combine the ability to recognize both human-triggered and system-triggered changes.
- For system-triggered changes, tool support to detect the conflicts and translate them in terms of model elements is required.
- Equally importantly, we need to be able to trace from lower models to upper ones to support reasoning and proper reactions—i.e. if the open system level detects that a service fails, traceability allows us to identify the affected stakeholders to inform them accordingly.
Adapting the Service in Response to Change

- Adaptation will be similar to synthesis in the sense that *modelling and win-win negotiations play an important part while keeping the product line scope.

- **Example:** Once the stakeholders affected by a service failure are identified, a negotiation may be carried out to assess the criticality of this failure, possible alternatives, etc.

- Alternative services and components that are available may be explored to see how they fulfill requirements and conflict with stakeholder goals.

- *Decision tables*, as shown later, capture variation points and can be used to inform engineering of possible and meaningful changes that do not violate stakeholder goals.
A Real World Example

- **Important considerations:** High scalability and high availability.

- **Difference from the integrated approach:** The customer or business stakeholder is known, so we only need one SR model.

- **Challenges:** We are unable to achieve all of the stakeholder’s wishes, so we must rely on external dependencies or other services at Hooli.

- It is difficult to fathom possible services already existing within the company that can serve our needs; we must dig deep.
A Result from the Initial Design/RE Phase (Architecture Prescription)
Service Discovery and Adding Details and Granularity to the Design (Solution Architecture)

Fig. 6. Excerpt of Solution Architecture Model.
Keys to Successful Requirements Engineering

- Communicate with service owners
- Figure out how to utilize dependencies
- Troubleshoot potential issues
- Anticipate potential issues

TABLE III

<table>
<thead>
<tr>
<th>Decision Variable</th>
<th>Question</th>
<th>Selection Type</th>
<th>Cardinality</th>
<th>Link to i* element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Credit</td>
<td>What is your preferred service for credit card transactions?</td>
<td>Set</td>
<td>1</td>
<td>Credit Card</td>
</tr>
<tr>
<td>Card</td>
<td></td>
<td>{CheatCard, Securitas}</td>
<td></td>
<td>Card</td>
</tr>
<tr>
<td>Service Provider</td>
<td></td>
<td></td>
<td></td>
<td>Service Provider</td>
</tr>
</tbody>
</table>
Postmortem of My Co-Op

- Accounting for all the highly granular and super specific business logic that your algorithm has to execute ...

- ... when all you have are generalities from the business stakeholders ...

- Anticipating and formulating what the business actually wants is tough.

- Synthesizing different dependencies together to solve your problems is also tough.

- Implementation introduces unexpected challenges.
Helpful Tips

- Be nimble and be ready for unexpected changes in the requirements or services or dependencies.
- Don’t be afraid to stray from the plan.
- However, always have an idea of and stick to satisfying the user’s needs.
- **Hooli specific:** Start with the customer, and the rest will follow.