Design Issues for Modeling Behavior of Distributed System

Harshada Dhande
IT department
Vidyalankar Institute of Technology
Mumbai University.
harshudhande@gmail.com

***Abstract*—There are several issues in the distributed system as scalability, flexibility, heterogeneity, security, abstraction Etc. By considering this issues we design a distributed system but currently using models does not satisfies design issues completely Hence by putting appropriate abstractions we can improve performance of distributed system. This paper describes the survey on these designing issues in modeling behavior of a system.**

I. INTRODUCTION

Many software engineering projects involve the automation of "real-world" systems, consisting of interacting entities that work together to perform specific tasks. These real-world systems often exhibit intrinsic qualities that parallel the inherent characteristics of distributed software systems. One of these qualities is concurrency. A real-world system possesses this quality if it includes more than one entity at any time, which most do. For example, consider a system for shipping parcel packages around the world. At any time, this system may include many physical entities (e.g., parcels, customers, and locations) and abstract concepts (e.g., rates, origins, and destinations.) Each of these can exist independently and behavior concurrently.

Distributed software systems provide true concurrency by disbursing objects over a network of computing resources. In contrast, centralized (single-processor) systems can only simulate concurrency. Another shared quality is partial failure. In real-world systems, a single entity, like a parcel or customer, can fail independent of the rest of the system. The same is true for distributed systems since individual computing resources in the network can fail or become unreachable. In centralized systems, failure is usually treated as an all-or-nothing condition. A third common quality is dynamic, incremental change. Real-world systems rarely require a complete shutdown to make changes in the way that its work is performed. In fact, change is often considered an integral part of a real-world system instead of an external process. Can you image a worldwide shipping company shutting down its entire operation every time an office needed to add a new piece of equipment or open up a new distribution channel? The same situation exists for distributed systems. Because of the impracticality of shutting down an entire distributed system, mechanisms for dynamic and incremental change are often built into the infrastructure. The similarities between real-world and distributed software systems, might lead one to conclude that an accurate conceptual model of the first could be easily transformed into a good design for the second. However, this is not true because of some subtle and not-so-subtle differences in what the models represent and because of additional design issues related to distributed systems.

II. CHALLENGES IN GOING FROM ANALYSIS TO DESIGN

In general, the process of going from analysis to design in software development has always involved mapping or transforming conceptual models. Models used during analysis, regardless of whether they are formal, informal or subconscious, aim to describe real-world systems from a problem-domain perspective. The concepts found in an analysis model should relate directly to concepts in the real-world system. Models used for software design, on the other hand, involve an additional level of abstraction and serve a different purpose.

They describe software concepts, like objects, structures, and processes, that only indirectly relate to concepts from the problem-domain. The purpose of a design model is to provide a blueprint for implementation and a framework for subsequent evolution of the system. It used in the maximal breach path algorithm, can built into the plane with randomly placed discrete set of points(sites). In 2D,diagram of the set of discrete sites partitions the plane into a set of convex polygon are closest to one site. This construction effectively produces polygon with edges that are equidistant from neighboring sites. Ideally, the conceptual distance between an analysis and a corresponding design should be kept to a minimum. This improves understandability, traceability, and maintainability. For distributed systems, however, there are several fundamental problems that tend to increase the conceptual distance and complicate the design process. Below is a summary of some of them.

A. Complex Mapping

The mapping between concepts in the analysis and those in the design is not one-to-one. A single real-world entity can be represented by many software objects. For example, to increase access time, a parcel-tracking system may include multiple software objects for a single parcel and distribute them across the network.

B. Fragmented Behaviour

A software object may represent only a portion of an entity described in the analysis. I will refer to such partial representations, as *object fragments* [Clyde-1993]. The object fragments that represent a specific real-world object do not have to be uniform, as long as they collectively encapsulate the required information and behavior. For example, in a client/server environment, client object fragments are often very lightweight and communicate with their server counterparts to provide users with the complete functionality.

C. Object Distribution

The distribution of software objects doesn't have to mirror the distribution of corresponding entities in the real-world. Just because a parcel is in Boston, doesn't mean that the software objects that represent it have to be in Boston. Decisions on how to distribute objects across system should consider other factors like performance, reliability, security, and fault tolerance.

D. Emergent communication play a major role

Distributed software systems involve many of different kinds of communications. Most of them, however, do not relate directly to communications in the problem domain. Some exist for house keeping reasons, like replica management, process and transaction synchronization, name resolution, and service binding. Others exist as a consequence of using a particular communication architecture. In either case, these emergent communications play a significant role in design, and therefore, cannot be left out of concept model.

E. Emergent Resource Sharing

Distribution can also introduce a significant amount of resource sharing that didn't exist in the analysis. For example, in the real-world parcel system, only one person is able to handling a given parcel at a time. In a distributed software system, many different users may be accessing and updating the software object(s) for that parcel at that time.

F. Transparency

To help make a distributed system more open, extensible, and fault tolerant, designers attempt to shield users from issues dealing with the actual location of an object (or service), concurrent access, replication, migration, scaling, and failures. This principle is called *transparency* [Coulouris-1994]. Unfortunately, techniques for achieving transparency further increase the conceptual distance between analysis and design.

III. Requirement for better conceptual model

Although some of the problems mentioned above are due to inadequate development tools for distributed systems, it is not likely that better, more sophisticated tools will make a difference. In fact, they might even make the problems worse. Inappropriate use of a new tool could accidently increase the emergent communications and resource sharing, and thus, further complicates the design. A fully integrated and minimized collection of development tools could substantially reduce the complexity of distributed systems design. But, in an open market with a wide diversity of platforms and with a constant infusion of new technology, such a development environment is not realistic. Our best hope is to improve our conceptual models for distributed software systems. Below are a few requirements, beyond those commonly found in existing conceptual models,

- Abstractions for mapping software objects (or object fragments) to entities described in the analysis.
- Abstractions for describing object fragmentation, especially for behavior.
- Better mechanisms for specializing objects and object fragments in ways that don't violate inherited semantics.
- Abstractions for specifying and constraining object distribution, replication, and migration.
- Leveled abstractions for dealing with emergent communications and resource sharing.
- Abstractions for proven patterns or idioms that can help designer achieve transparency and other desirable properties.

These proposed abstractions should give software engineer control over design decisions, but allow them to hide the details when they are not pertinent to a particular discussion.

IV. CONCLUSION

Distributed systems design is unnecessarily complex because our current conceptual models do not provide the right kinds of abstractions. By adding appropriate abstraction to our models, we can also reduce the conceptual distance between analysis and design.

REFERENCES

[1][Clyde-1993] Clyde, S., "Object Mitosis: A Systematic Approach To Splitting Objects Across Subsystems", International Workshop on Object Orientation and Operating Systems, December, 1993

[2][Coulouris-1994] Coulouris, G. F., J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, Second Edition, Addison Wesley, 1994