Digital Twins as the Keystone of the Design of Distributed Systems

Luc Hogie

▶ **To cite this version:**

Luc Hogie. Digital Twins as the Keystone of the Design of Distributed Systems. CNRS. 2024. hal-04878631

HAL Id: hal-04878631
https://hal.science/hal-04878631v1

Submitted on 10 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Digital Twins as the Keystone of the Design of Distributed Systems

Luc Hogie
luc.hogie@cnrs.fr
ORCID: 0000-0001-8044-5672
CNRS/I3S/Inria/Université Côte d'Azur

Summer 2024

Abstract

The design of distributed systems has relied of several strategies to represent and provide access to remote components: from explicit message-passing, to transparent invocation of remote functions, stubs, Web-oriented APIs, etc; new approaches being proposed as technology evolves. This paper discusses the pros and cons of these strategies, and it proposes a new approach called Idawi Digital Twining System (IDST) for the design of distributed applications, which relies on the concept of *digital twin*. It explains how IDST takes advantages of the intrinsic properties of digital twins, and which are its positive impacts on (the development of) distributed applications. Finally its describes the Idawi framework for distributed computing, which serves as an Open Source Java reference implementation for IDST.

1 Introduction

Distributed/parallel computing is inherently difficult, as communications and concurrency dramatically increase the complexity of design of systems as well as the opportunity of failures. Worse, concurrency problems are often counter-intuitive, and the distributed nature of the computing environment makes them harder to detect, understand and solve. Because of this, the availability of libraries/frameworks for distributed computing is of paramount importance.

2 Existing tools

Thanks to its simplicity and robustness, MPI remains—even today—the most popular framework for scientific computing. ProActive [5], MPI, JXTA, JMS (Jakarta Messaging API), ActiveMQ [8], JGroups [2], RMI, Akka, JavaCà&Là

(JCL) [7], GoPrime [4], ParallelTheater [13], ActorEdge [1], and EmbJXTA-Chord [3], are either commercial/Research tools coming with effective solutions to some of the problems of distributed computing. In the contexts of pervasive/edge/mobile computing, VANETs, the IoT, etc, recent works consider decentralization.

3 The problem: dealing with remote entities

In a distributed system, entities (components, objects, functions, etc) live in different address spaces. For this reason they cannot refer to each other via (process-scoped) pointers/references: a distributed referencing system is needed.

In many systems, remote entities are represented by numerical or textual identifiers. In this approach there is no way for the compiler to assist the developer and verify its source code, as the numerical of textual IDs representing remote things have no semantics a compiler can manipulate: they are just values. This approach has the following pros and cons.

On the one hand, a significant advantage of this approach is that it makes distribution explicit: remote calls can be clearly seen in the source code of applications. The developer can then have a clear idea of when his application will use the network, allowing him to minimize as much as possible communications, which are a common bottleneck in distributed applications.

On the other hand, the natural—elegant—constructs of the programming language cannot be used to refer to remote things. This tends to obfuscate the source code to make the code prone to bugs. A naming system based on types is always desirable as it offers much more security and consistency of the code.

Object-oriented programming proposes an elegant solution to the problem of referring to remote entities, by associating them to specific types, and forcing their local counterpart to be of this same type: both the remote entity and its local *stubs* follow the same specification (they extend the same class). The code for stubs, which is generated automatically by a preprocessor, implements bidirectional communication with the remote object. The business implementation must be provided by the developer. This approach has interesting properties. First, it takes full advantage of OO designs (relying on abstraction, polymorphism, inheritance, and encapsulation), ensuring enhanced readability and maintainability, extensibility, code reuse, and reduced development time. Second, it provides distribution transparency. A stub can then be used as a placeholder for the remote object it refers to. Consequently, algorithms manipulate stubs just like they would do with business objects. Transparency seems highly profitable at first glance, but it actually exhibit two major drawbacks:

First, by hiding distribution, it hides its inherent cost. When invoking a method on a stub, there is no indication whatsoever that the invocation will not only entail a jumps in memory like this happens for any method invocation, but also and bi-directional synchronous network communication. From a more technical perspective, while invoking a local implementation take just a few nanoseconds, adding a network communication on the fastest local networks

today is more of the order of a millisecond (in the order of 10^3 slower). This performance problem is not visible in the source code. Transparency then tends to produce too high-level and largely inefficient code.

Second, because the functional interface is the same for stubs and business objects, the programmer is tempted to think they behave the same. But there exists a major difference in terms of behavior. As a matter of fact, when invoking the business object *locally* (via its native reference), mutable parameters are passed to methods as references, and immutable ones are passed by value. When invoking business methods via a stub, (de)serialization enters the scene, and parameters get all passed by value, be they mutable or immutable. This also applies to return values. Unfortunately, this difference of behavior is not reflected by the source code. The developer must be aware of the nature (stub or business object) of the instance it manipulates, so as to treat parameters and return values differently. An intuitive solution to this problem is to dynamically/transparently create stubs to all mutable objects, and to transport these stubs instead of the values of objects. Remote algorithms would then deal with stubs of objects living on the same node/process from which the execution request originates. Unfortunately doing this worsens the impact of transparency, increases the amount of communication, thereby dramatically reducing the global performance of applications.

4 The digital twin approach

4.1 Description

In this paper, we propose a novel design, called Idawi Digital Twining System (IDTS), for distributed systems, which relies on the concept of *digital twin* (DT). A DT is a model of a physical object. It evolves alongside its physical counterpart, and it offers the ability to obtain information on it, and to simulate it.

The main idea in IDTS is to represent, as well as refer to, remote so-called "physical" objects as DTs. IDTS takes advantage of the very nature of the physical object it considers: software objects. These are of the same nature as DTs: they both are made of code. IDTS defines that a digital twin has the same code as the real object. They only differ in the data they carry, and the role they play. When queried, an object declared as the digital twin of another object, can do several things:

- it can act like a stub, forwarding the query to the real object (optionally caching the result locally);
- or if it has the necessary data and its host node has enough resources available, it can execute the business code (optionally reporting its action to the real object, and caching the result locally);
- or it can simulate the behavior of the real object (optionally reporting its action to the real object, and caching the result locally);

- or it can return a previously cached value.

In addition to its natural conformance to the OO approach, as DT can serve as a placeholder for the real object (it can actually even become the real object if it obtains the necessary data), IDTS has a number of desirable impacts on distributed systems, not found in existing designs:

- It makes them simpler (both in design and source code) by making local and remote objects of the very same nature;
- It enhance their flexibility by natively providing query delegation, result caching, and simulation;
- By enabling each node to simulate the distributed system (not just the network), it can help applications to improve the management of resources, hence their performance.

4.2 An use case: message broadcasting

In order to illustrate the advantages of being able to simulate locally in this context, please consider the following use case: node needs to broadcast a message into the network. To this purpose it needs to evaluating the best diffusion strategy, which is a multi-objective problem aiming at (among others):

- minimizing network congestion;
- minimizing energy consumption;
- maximizing delivery rate.

In order to broadcast a message, every node has a set of so-called *broadcast services*. Each of them comes with its own set of parameters. If we assume that only one broadcasting algorithm is going to be used, mathematical analysis can compute a Pareto front, which will permit the calibration of the values of each parameter. But if we relax that unrealistic assumption, maths become way too complex, and simulation becomes the only way to go. In a DT-based system, the node will be able to simulate the broadcast process on its *memory-local network of DTs*. In this simulation, the full diversity of behaviors found in the physical system can be considered for further decisions on what to do..

Just like link-state routing protocols do, physical nodes disseminate topology information across the network, but not only. Indeed "routing" is not our sole concern. A framework for distributed computing must offer services for resource/services discovery, among others. Then physical nodes must also send structural and system information, like the services they offer, their computational/storage capabilities, their current load, etc. This information can then be used to update individual physical nodes to update their local digital twins of remote nodes so as their local view of their network environment remains as accurate as possible. Accurate views can then be cleverly used by applications to distribute at best.

5 Reference implementation: the Idawi framework

IDAWI [11][10][12] is a framework for decentralized computing on large heterogeneous dynamic networks. It aims at providing applications built atop of it with appropriate communication and computation models, as well crucial services like messaging, routing, service discovery, provisioning, deployment, etc. The initial motivation for the design and implementation of a new framework was to meet the technical requirements of the scientific experimentation of the COATI/I3S/Inria/CNRS Research group (2010-2020) in the contexts of its Research projects on graph algorithms [14], networking [6], decentralized protocols [9], etc. More specifically we needed software services for the deployment of workers, their monitoring and the control of their jobs in progress.

In 2021, our production code turned into a Research platform when we worked at making it effective at large scale (i.e. when a large number of computational nodes are involved). In this context, the difficulties lied to node failures, churn, mobility and heterogeneity arose naturally. In particular we considered heterogeneity anywhere it can be encountered: on the hardware of nodes (RAM, CPUs, disks, etc), and on the communications mediums (wired, wireless). The latter aspect impacts deeply the communication strategies for using wireless technologies inevitably leads to the presence of directed links into the communication graph.

IDAWI is Open Source, released under Apache v2 licence and its source code is publicly available at:

`https://i3s.univ-cotedazur.fr/~hogie/idawi/`

The very nature of the network we consider (i.e. large, dynamic and heterogeneous) relaxes most of the assumptions made by other frameworks. This led us to rethink the design of most of our initial code. In this process, as much as possible we look at the state of the Art of existing tools, and try to reuse concepts, ideas, and codes.

5.1 What a component is: application model

The application model of IDAWI relies on the Service Component Architecture (SCA) style of programming. SCA defines that functionality is brought by services/endpoints in components. This is true at both business and system levels. In IDAWI, components are *atomic*, meaning that a single component cannot be split on several nodes. Distribution is brought by specific services dedicated to network messaging: *transport* services (OSI layer 2) and *routing* services (OSI layer 4).

Such a *structuring model* forces distributed applications to conform to a certain organization defined by a specific OO model, which ensures *consistency* of source codes, significantly reduces the risk of design errors (for most design work is into the framework), and it enables the addition of high-level functionalities such as *deployment* and *service discovery*.

5.2 How do components communicate: communication/-computation model

While the application architecture conforms to the OO approach, and despite the OO philosophy would foster the use of remote method invocation (like it is done in CORBA, ProActive, etc), for the sake of flexibility we opted for an asynchronous message-passing communication scheme. Upon reception, a message is delivered into its target queue, unless its content is an endpoint execution request. Each time it is invoked, an endpoint is provided with a queue it will consume. In turn, along its execution, an endpoint will optionally generate new messages.

Each endpoint execution is assigned to an available thread from a pool. The number of threads available to simultaneously running endpoints is not limited, but they are reused as much as possible, so as to minimize thread overhead.

The coupled communication/computation models have been designed this way so as to maximize flexibility: multicast makes it *collective* by nature; queues enable both reactive (asynchronous), imperative (synchronous) programming, as well as the implementation of many workflows.

5.3 Digital twins

As said hereinbefore Unlike stubs and business objects which constitute two different implementations of the same business abstraction, components and digital twins are essentially the same thing. They expose the same business services whose endpoints have the exact same implementation. Within the implementation, a component is considered to be a DT if it has a *digital twin service*. Without this service, a component is considered to be a real object and will always execute its business code. When this service is activated, the component will behave like a DT, as described in Section 4.1. This behavior can be further driven by business-specific information carried by messages.

6 Conclusion

The design of frameworks for distributed computing has been changing a lot during the past 3 decades. OO modeling, SOA, and microservices architectures have brought new concepts and possibilities which proved highly beneficial to the development of distributed applications. We believe that, on top of these paradigms, the notion of digital twin can play a major role in the design of future distributed applications, and in particular at framework-level. Using digital twins as placeholders for remote entities, along with message-passing, cumulates the advantages of OO designs while getting rid of (some of) its drawbacks. In order to demonstrate the implementability and effectiveness of this approach from a programmatic point of view, we implemented it at the core of the Idawi framework.

Acknowledgments I am grateful to all of those with whom I have had the pleasure to work since the beginning of this project, and especially to Master's students Valentin Mascaro and Sami Joudet.

References

[1] Austin Aske and Xinghui Zhao. An actor-based framework for edge computing. In Ashiq Anjum, Alan Sill, Geoffrey C. Fox, and Yong Chen, editors, *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, pages 199–200. ACM, 2017.

[2] Bela Ban. Adding group communication to java in a non-intrusive way using the ensemble toolkit. Technical report, Citeseer, 1997.

[3] Filippo Battaglia and Lucia Lo Bello. A novel jxta-based architecture for implementing heterogenous networks of things. *Comput. Commun.*, 116:35–62, 2018.

[4] Mauro Caporuscio, Vincenzo Grassi, Moreno Marzolla, and Raffaela Mirandola. Goprime: A fully decentralized middleware for utility-aware service assembly. *IEEE Trans. Software Eng.*, 42(2):136–152, 2016.

[5] Denis Caromel, Alexandre di Costanzo, and Clément Mathieu. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Comput.*, 33(4-5):275–288, 2007.

[6] David Coudert, Luc Hogie, Aurélien Lancin, Dimitri Papadimitriou, Stéphane Pérennes, and Issam Tahiri. Feasibility study on distributed simulations of BGP. *CoRR*, abs/1304.4750, 2013.

[7] Leonardo de Souza Cimino, José Estevão Eugênio de Resende, Lucas Henrique Moreira Silva, Samuel Queiroz Souza Rocha, Matheus de Oliveira Correia, Guilherme Souza Monteiro, Gabriel Nata de Souza Fernandes, Renan da Silva Moreira, Junior Guilherme de Silva, Matheus Inácio Batista Santos, André Luiz Lins de Aquino, André Luís Barroso Almeida, and Joubert de Castro Lima. A middleware solution for integrating and exploring iot and HPC capabilities. *Softw. Pract. Exp.*, 49(4):584–616, 2019.

[8] Nicolas Estrada and Hernán Astudillo. Comparing scalability of message queue system: Zeromq vs rabbitmq. In *2015 Latin American Computing Conference, CLEI 2015, Arequipa, Peru, October 19-23, 2015*, pages 1–6. IEEE, 2015.

[9] Luc Hogie. *Mobile Ad Hoc Networks: Modelling, Simulation and Broadcast-based Applications. (Réseaux Mobile Ad hoc : modélisation, simulation et applications de diffusion)*. PhD thesis, University of Luxembourg, 2007.

[10] Luc Hogie. IDAWI: a decentralised middleware for achieving the full potential of the iot, the fog, and other difficult computing environments. In *Proceedings of MiddleWedge 2022 ACM International workshop on middleware for the Edge. Collocated with ACM/IFIP/USENIX Middleware 2022, Québec, Canada*. ACM, 2022.

[11] Luc Hogie. A service-oriented middleware enabling decentralised deployment in mobile multihop networks. In *Proceedings of FMCIoT 2022 International Workshop on Architectures for Future Mobile Computing and Internet of Things. Collocated with 20th International Conference on Service-Oriented Computing (ICSOC 2022), Sevilla, Spain*. LNCS, 2022. To be published.

[12] Luc Hogie. A Decentralized Web Service Infrastructure for the Interoperability of Applications in Multihop Dynamic Networks. In *CIoT 2023 - 6th Conference on Cloud and Internet of Things*, pages 211–218, Lisbon, Portugal, March 2023. DNAC, IEEE.

[13] Libero Nigro. Parallel theatre: An actor framework in java for high performance computing. *Simul. Model. Pract. Theory*, 106:102189, 2021.

[14] Thibaud Trolliet, Nathann Cohen, Frédéric Giroire, Luc Hogie, and Stéphane Pérennes. Interest clustering coefficient: a new metric for directed networks like twitter. *J. Complex Networks*, 10(1), 2021.