Notification Oriented Paradigm for Distributed Systems

Wagner R. M. Barretto[1], Ana Cristina B. Kochem Vendramin[1], Jean M. Simão[1]

[1]Informatics Department (DAINF)
Federal University of Technology - Paraná (UTFPR)

wagner.rezende@gmail.com, cristina@dainf.ct.utfpr.edu.br,
jeansimao@utfpr.edu.br

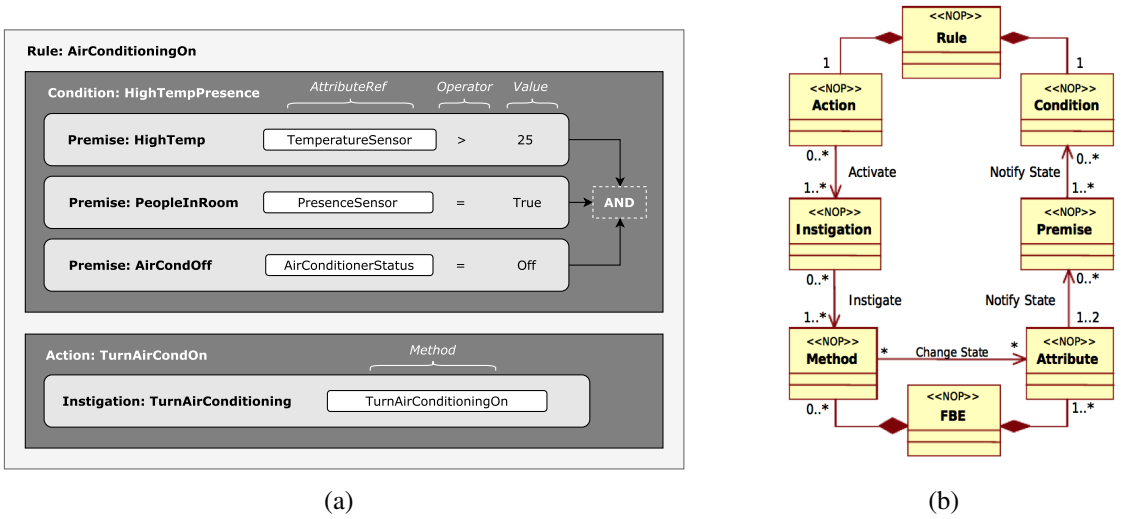
Abstract. *Notification Oriented Paradigm (NOP) has been proposed as a new way to design software that is more efficient, decoupled, and with better performance than other paradigms. NOP is built based on a well-defined set of entities that interact by means of notifications. The way those entities are designed enables a declarative and rule-based programming model that is suitable for distributed systems. This paper introduces a method to write distributed NOP programs that maintains the same characteristics of performance and cohesion that its local counterpart has. The method is presented with two case studies that have their design and performance compared to equivalent programs written with traditional models and paradigms. The results show that distributed NOP programs behave correctly and, beyond the distribution, present similar benefits as their single instance counterparts.*

1. Introduction

Recently, a new programming paradigm, called Notification Oriented Paradigm (NOP), was proposed in order to solve software development issues in terms of ease composition of optimized and distributable code [Simão and Stadzisz 2009]. The NOP basis was initially proposed by J. M. Simão as a manufacturing discrete-control solution [Simão 2005]. This solution was evolved as a general discrete-control solution and then as a new inference-engine solution, finally achieving the form of a new programming paradigm [Simão et al. 2012]. The essence of NOP is its inference process based on small, smart, and decoupled collaborative factual-executional entities and logical-causal entities that interact by means of precise notifications. NOP enables the development and execution of software in a way that keeps the main advantages of both declarative programming (i.e. higher causal abstraction and organization) and imperative programming (i.e. reusability, flexibility and structural abstraction). [Simão and Stadzisz 2009].

Imperative programming paradigms like Procedural or Object Oriented motivated the creation of distributed programming paradigms counterparts like RPC and Distributed Objects [Kendall et al. 1994]. On the other hand, declarative and rule-based paradigms proved difficult to be adapted to a distributed environment due to issues such as the high coupling between entities [Simão and Stadzisz 2009]. NOP solves most of those issues while still maintaining a declarative rule-based programming model. This makes NOP a suitable choice for distributed systems experiments.

Supported by NOP's decoupled structure, its creators claim that NOP programs can be seamlessly distributed across processes or even networks. However, by this date little progress had been made to demonstrate NOP in a distributed systems context and



(a) (b)

Figure 1. *Rule* example (a) and NOP UML class diagram (b) [Simão et al. 2012].

none has been published yet. This paper presents a method to write distributed NOP programs that still maintains the main benefits and properties of the paradigm. This paper also demonstrates the use of NOP in distributed systems by means of two case studies. Those had their results analyzed in terms of correctness and computational resource usage.

The paper is structured as follows: Section 2 presents a background about NOP and its main characteristics. Section 3 discusses how NOP would fit in a distributed systems context and proposes a method for writing distributed NOP programs. Section 4 presents two case studies that were conducted using the proposed method and presents their results. Section 5 presents related and future work. Finally, Section 6 presents concluding remarks.

2. Notification Oriented Paradigm (NOP)

The Notification Oriented Paradigm (NOP) introduces a new concept to conceive, construct, and execute software applications. NOP is based upon the concept of small, smart, and decoupled entities that collaborate by means of precise notifications to carry out the software inference [Simão and Stadzisz 2009]. This allows enhancing software applications performance and potentially makes easier to compose software, both non-distributed and distributed ones [Simão et al. 2012].

2.1. NOP Structural View

NOP causal expressions are represented by common causal rules, which are naturally understood by programmers of current paradigms. However, each rule is technically enclosed in a special computational-entity called *Rule*. An example of *Rule* Entity content is illustrated in Figure 1a. This *Rule* structures and infers the causal knowledge of an application that controls an air conditioning system based on information provided by various sensors.

Structurally, a *Rule* has two parts, namely a *Condition* and an *Action*, as illustrated by the UML class diagram in Figure 1b. Both parts are entities that work together to

handle the causal knowledge of the *Rule*. The *Condition* is the decisional, or logical-causal part, whereas the *Action* is the execution part of the *Rule*.

NOP factual elements are represented by a special type of entity called "Fact Base Element" (*FBE*). An *FBE* includes a set of attributes. Each attribute is represented by another special type of entity called *Attribute*, such as *TemperatureSensor* and *PresenceSensor* (see Figure 1a).

Attributes states are evaluated in the *Conditions* of *Rules* by associated entities called *Premises*. In the example, the *Condition* of the *Rule* is associated to three *Premises*, which verify the state of *FBE Attributes* as follows: (a) Is the Temperature higher than 25 degrees? (b) Are there people in the room? (c) Is the air conditioner turned on?

When each *Premise* of a *Rule Condition* is in true state, which is concluded by means of a given inference process, the *Rule* becomes true and can activate its *Action* that is composed of special-entities called *Instigations*. In the considered *Rule*, the *Action* contains only one *Instigation* that turns the air conditioning system on.

In fact, *Instigations* are linked to and instigate the execution of *Method*s, which are another special-entity of *FBE*. Each *Method* allows executing services of its *FBE*. Generally, the call of an *FBE Method* changes one or more *FBE Attribute* states, feeding the inference process.

2.2. NOP Inference Mechanism

The inference mechanism of NOP is innovative since the *Rules* have their inference carried out by reactive collaboration of its notifier entities [Simão and Stadzisz 2009]. The collaboration happens as follows: for each change in an *Attribute* state of an *FBE*, the state evaluation occurs only in the related and pertinent *Premises* and then in related and pertinent *Conditions* of *Rules* by means of punctual notifications between them.

In order to explain the Notification Oriented Inference, it is necessary to explain the *Premise* structure. Each *Premise* represents a Boolean value about one or even two *Attribute* states, which justify its structure: (i) a reference to an *Attribute* discrete value, named as Reference, that is received by notification; (ii) a logical operator, named as Operator; and (iii) another value named as Value that can be a constant or even a discrete value of other referenced *Attribute*.

A *Premise* executes a logical calculation when it receives notification of one or even two *Attributes* (i.e. Reference and Value). This calculation is carried out by comparing the Reference with the Value, via the Operator. In a similar manner, a *Premise* collaborates with the causal evaluation of a *Condition*. If the Boolean value of a notified *Premise* is changed, then this *Premise* notifies the related *Condition* set.

Therefore, each notified *Condition* calculates their Boolean value by the conjunction of *Premises* values. When all *Premises* of a *Condition* are satisfied, the *Condition* is also satisfied and notifies the respective *Rule* to execute.

An important point about NOP collaborative entities is that each notifier (i.e. *Attributes*) registers its client (i.e. *Premises*) in its creation time. For instance, when a *Premise* is created and makes reference to an *Attribute*, this attribute automatically includes that *Premise* in its internal set of entities to be notified when its state changes.

3. Distributed Systems and NOP

Many different distributed computing models were proposed over the last decades (message passing in the 1970's [Cook 1980], remote procedures in the 1980's [Birrell and Nelson 1984], distributed objects in the 1990's and web services in the 2000's). Most of the efforts in creating those models were motivated by the need to match them more closely to the programming paradigm in vogue at the time [Kendall et al. 1994]. With NOP, it is natural that new distributed programming techniques emerge to address distributed systems challenges in a way that remains true to NOP's fundamentals.

NOP creators claim that NOP-based systems can be fully distributed among processes and nodes in a computer network [Simão et al. 2012]. This claim is backed by the highly decoupled object model that NOP is based on and the notion of *Distributed Objects* popularized by middlewares such as CORBA and Java RMI. In NOP creators vision, an entire NOP program would have its entities distributed across a set of network nodes. These entities would then collaborate in a distributed object fashion to perform the task of a NOP program.

NOP's parallelism capabilities had been demonstrated in single instance systems [Belmonte et al. 2016]. However, distribution of computations across processes in a same machine is fundamentally different than distribution across a set of machines connected though a network [Kendall et al. 1994]. The main differences are that memory access is not shared and network environments introduce the possibility of partial failures. Techniques like distributed objects, which helped to foment NOP's distribution capabilities claims, tried to abstract the network complexities from the programmer of an object oriented system by treating remote objects as if they were local. Even though the distributed objects approach to distribute NOP programs is possible, it would inherit all problems associated with the technique.

We argue that NOP distribution efforts should not try to partition programs in a way that each node holds different types of NOP entities that must act together to perform a task. This would generate programs difficult to reason about and tolerate partial failures. Instead, we propose that complete NOP processes should be able to communicate with each other in a way that is adherent to NOP's rule-based mechanisms. By "complete NOP processes" we mean programs with well-defined responsibilities that make use of all of NOP's entities. To achieve this, a careful consideration of NOP's entity model should be made to identify points where distribution is advantageous. Afterwards the model should be extended to support distribution. In all cases, distribution points should be explicitly defined by the developer. This way all the peculiarities of network programming like latency, non-shared memory access, and concurrency can be more easily taken into consideration.

In this work we present an extension to NOP that makes it possible to NOP processes to communicate with each other while still maintaining a pure declarative rule-based programming model. This is achieved by the introduction of the concept of *Distributed Attributes*, which are a specialized type of NOP *Attribute* that is able to share its state with other NOP processes.

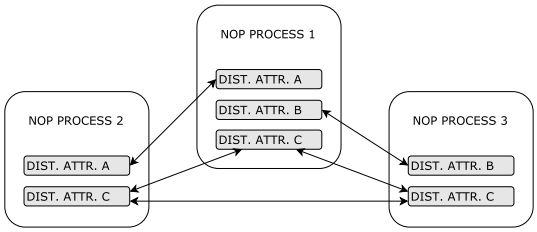


Figure 2. *Distributed Attribute* Model Representation

3.1. Distributed Attribute Model

A *Distributed Attribute* is a new type of NOP *Attribute* that can be shared among many NOP processes. A *Distributed Attribute* interact with other NOP's entities in the same way as a normal NOP *Attribute*; it can be part of *Premises* and has its state changed by *Methods*. However, there are two main differences between a distributed attribute and a normal one. The first is that whenever a *Distributed Attribute* has its state changed, it publishes a notification message with the new state to the network. The second is that upon initialization, the *Distributed Attribute* sets up a network listener that reads messages with new states. Whenever a message arrives, the *Distributed Attribute* updates itself with the new state.

The idea is that a *Distributed Attribute* can have its state replicated among any number of distinct NOP processes connected by the same network. An example scenario illustrated by Figure 2 shows three NOP processes sharing the *Distributed Attributes* A, B, and C. We can see that the *Distributed Attribute* C is shared among all three processes, while *Distributed Attributes* A and B are only shared between two processes.

The way the *Distributed Attribute* is designed allows a multi-directional state transfer between any number of attributes. This can certainly generate concurrency problems that would lead to inconsistent states between programs. This is a classic distributed transaction problem with solutions [Gray 1978, Lamport 1998] and limitations [Fischer et al. 1985]. We do not impose any specific solution and leave such details to the implementation.

4. Distributed Attribute Model Validation

To validate the *Distributed Attribute*, an implementation of the proposed model was made and two case study programs were built on top of it. The *Distributed Attribute* model was implemented in the Java NOP materialization. The first case study is an electronic gate simulator. It is a usual problem that has been a case study for NOP application in some related works [Xavier 2014]. The second is the distributed transaction protocol Two Phase Commit [Gray 1978].

The two case study programs were compared to applications solving the same problem in the traditional imperative object oriented paradigm using remote procedure calls [Birrell and Nelson 1984] as the distributed programming model. Both paradigms are widely used in industrial and scientific applications alike. The results were verified in terms of correctness of the applications and by the network efficiency, measured by the number of messages exchanged in both applications. The purpose of the verification is to measure the overhead generated by NOP in a distributed setting.

4.1. Distributed Attribute Implementation

NOP's usage has been made primarily by the use of *NOP Frameworks*. These are implementations of NOP's concepts in different programming languages that provide an *Application Programming Interface* (API) to programmers. The first NOP framework was created in C++ and, subsequently, ported to Java and C# languages. For the implementation of the *Distributed Attribute*, we chose the Java Framework.

The *Distributed Attribute* class was implemented by extending the *Attribute* class already present in the framework. This new class provides a constructor where a network address is required as a parameter. Upon initialization, an instance of the *Distributed Attribute* class starts a network listener on the given address in a separate thread. Whenever the state of the *Attribute* is changed, it serializes its state and sends it to the provided network address. Whenever a notification message arrives, it updates its internal state.

The network communication was implemented by using UDP multicast. Every *Distributed Attribute* registers itself in a unique multicast address and listens to messages sent to this address for the entire program execution. When a new message is received, the *Attribute* updates its own state and propagate notifications to the *Premises* running in the same local process. When an *Attribute* has its state changed by a local NOP *Method*, it sends its new state over its multicast group. The solution ensures that participants only receive messages regarding *Attributes* that they explicitly registered interest. The implementation can be easily modified to support TCP broadcast or even TCP unicast. Multicast was chosen to guarantee maximum message delivery efficiency in local area networks.

4.2. Case Study 1 - Electronic Gate

The first case study comprises a system that simulates the software that runs in an electronic gate and its remote controls. The system is comprised of two distinct types of processes: one that runs in the gate and one that runs in the remote controls. The system has a basic functionality of changing the gate state every time one of the remote controls has its unique button pressed. The gate states follow the state diagram shown in Figure 3.

The NOP implementation of the remote control software is comprised of an *FBE* with a single *Distributed Attribute buttonPressed* with two possible values representing the state of the button. No other NOP entities were necessary for this program. The NOP implementation of the gate software is comprised of one *FBE* with two *Attributes*, one being the distributed *buttonPressed* that is shared with the remote control program, and a non-distributed one representing the state of the gate. The gate program also has 6 *Rules* and *Conditions* (one for each possible gate state), three *Actions*, *Instigations*, and *Methods* (to stop, open, and close the gate) and seven *Premises* (button pressed, gate opened, gate closed, gate stopped opening, gate stopped closing, gate opening, gate closing).

One interesting property of this particular implementation is that it can handle multiple instances of both programs running concurrently and still behave correctly. This would allow for example a redundant gate control operation to handle eventual failures.

The RPC implementation for the electronic gate system is also comprised of two types of processes, one for the remote control(s) and one for the gate. The Gate program exposes an interface with a single method *changeState*. The remote control program acts

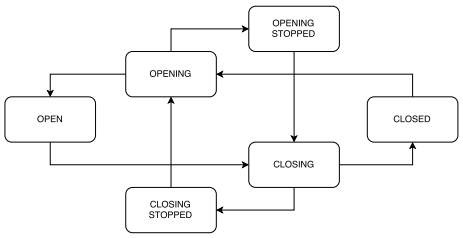


Figure 3. State diagram for the electronic gate

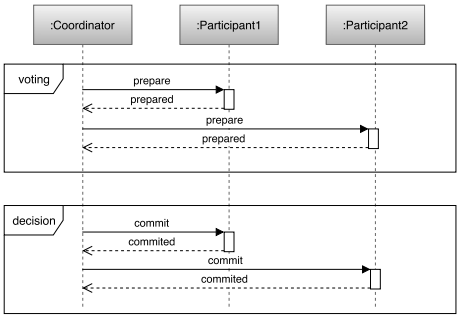


Figure 4. Two phase commit sequence diagram

as a client to that interface. Every time the remote control has its button pressed, a remote invocation is made to the gate's program exposed method, changing its state accordingly.

The results were measured in terms of the quantity of messages exchanged by the two programs. The distributed NOP implementation requires $2N$ messages and the RPC N messages, where N is the amount of times the remote control button is pressed. The increased number of messages necessary in the distributed NOP implementation is due to the way Notification works in NOP. Every time an *Attribute* state changes, the *Premises* interested in that *Attribute* have to be notified. As the *Distributed Attribute* passes for two state transitions in single press (NOTPRESSED $\rightarrow$ PRESSED $\rightarrow$ NOTPRESSED), at least two messages have to be sent over the network.

It is worth mentioning that traditional RPC implementations do not support one procedure call to be made in multiple hosts with the same method exposed. Therefore, the redundancy feature supported by the distributed NOP implementation would require more messages to be exchanged in the RPC implementation.

4.3. Case Study 2 - Two-Phase Commit

The two-phase commit is a protocol that enables atomic distributed transactions in a system with an arbitrary number of participants. It achieves that by using a coordination mechanism where one single participant coordinates a voting and a decision phase. In the voting phase all the participants vote on whether or not they could process the transaction. In this phase all the participants try to persist the transaction value in an intermediary storage. In the decision phase the coordinator gathers all the votes, decides if the transaction must commit or abort and communicates the decision to all the participants. A sequence diagram to commit a transaction is shown in Figure 4, but implementations can vary.

The NOP implementation for the two-phase commit involves two programs, one for the coordinator and one for the participants or voters. All the communication between the coordinator and the participants occurs by means of *Distributed Attributes*. Each participant has one *Distributed Attribute* in an unique address to communicate their vote. The coordinator has to know the address of all *Attributes* before it can begin a new transaction. The coordinator maintains *Distributed Attributes* shared with all the participants to indicate the content of a transaction, the transaction phase, and the final decision. The transition from the voting phase to the decision phase is triggered by a NOP *Rule* that runs on the coordinator process. This *Rule* is activated when all the participants vote.

Immediately after the voting phase, the coordinator issues a decision in the form of a *Distributed Attribute* state change. This *Attribute* triggers *Rules* to send commit or abort in the participants.

To handle multiple concurrent transactions, the NOP implementation has to use $N*M$ addresses where N is the number of participants and M is the number of concurrent transactions. For some applications this might be considered a drawback.

The RPC implementation of the two-phase commit protocol involves interfaces with two exposed methods for the coordinator (*haveCommited* and *getDecision*), and three for the participants (*canCommit*, *doCommit*, and *doAbort*) [Coulouris et al. 2011]. This particular implementation behaves in a way that participants never communicate state changes in a proactive way. This behavior is the opposite of a NOP application behavior, where state changes are instantly communicated to the interested resources by means of notifications.

As the two phase commit is a protocol designed to tolerate various levels of successive failures, the best case scenario is used to measure performance. Considering N participants in a distributed transaction, the distributed NOP implementation requires at least $N + 3$ messages to complete one transaction, the RPC implementation requires $3N$ messages. The advantage in NOP is due to the communication strategy and NOP's proactive notification mechanism.

4.4. Results and Discussion

From the correctness point of view, both distributed NOP programs behaved properly. The distributed NOP programs produced the same output as their RPC-based counterparts. From the performance point of view, the two case studies performed different. The distributed NOP electronic gate program exchanged more messages between the processes than its RPC-based counterpart. On the other hand, the distributed NOP two-phase commit exchanged less messages than its RPC-based counterpart.

The electronic gate is an example of a program that does not benefit from the distributed attribute strategy. This scenario occurs because of the way *Methods* interact with *Attributes* in NOP. For every state change of a distributed attribute A in a process $P1$ that triggers a *Method* in process $P2$, and this *Method* changes again the state of attribute A, at least two messages are needed. In applications where network efficiency is critical, this pattern has to be avoided whenever possible.

The two-phase commit case study showed that a rule-based program using notifications can be more efficient than a distributed algorithm based in remote calls. It is important to notice that the two-phase commit protocol can be implemented with a low level message passing multicast approach that would have the same performance as the NOP example. The main point here is that NOP does not add any overhead to the protocol like RPC does, in fact NOP enables a declarative implementation to have the same performance achieved by a low level message passing implementation.

Apart from network resource usage, all the performance benefits from NOP are carried to the distributed attribute implementation. Running a NOP program in a distributed way does not imply in extra logical evaluations being computed.

5. Related and Future Work

Although the subject of declarative and rule-based distributed programming had some contributions in the past, it still remains an open research area. Among the works that made progress are *Netlog* [Grumbach and Wang 2010], a rule-based language created to describe network protocols in a declarative manner. Similarly, *Dedalus* [Alvaro et al. 2011] is a declarative language designed to specify distributed systems. An extension of *Dedalus*, *Distributed State Machine* [Lobo et al. 2012] was introduced to address some of *Dedalus* limitations regarding time management and improve on the verification of programs created with the language. All those languages are based on *Datalog* [Ullman 1984] and have a focus on the declarative design of network protocols. The use cases presented for those languages are very different from the ones NOP has been applied to. For this reason a direct comparison is not straight forward and is let for future work.

As regards to other possible future work, developing more application examples would help to further validate the applicability of the model. The comparison of NOP and event-oriented paradigms was subject of recent work [Xavier 2014]. Comparing the *Distributed Attribute* model with distributed event-oriented frameworks like the *Actors Model* [Agha 1985] would bring more perspective to the comparisons and the model itself.

6. Conclusion

In this paper, we demonstrated that NOP can be applied to solve distributed systems problems. The low coupling between NOPs entities enables NOP programs to act in collaborative way in a distributed system context.

This paper introduced the *Distributed Attribute* concept, a new type of NOP *Attribute* that can coexist in multiple NOP programs. The *Distributed Attribute* enabled NOP processes to communicate with each other in a concise way while keeping the distribution explicit to developers. Its implementation in the NOP Java framework and the two case studies presented practical usages of distributed NOP applications.

NOP systems written using *Distributed Attributes* behaved correctly and produced cohesive programs with well-established responsibilities that communicated seamlessly in a distributed environment. The programs presented as case studies showed that the use of rule-based programming paradigms like NOP are perfectly viable for building distributed applications. The network efficiency of distributed NOP programs when compared with well-established distributed programming models showed that the declarative aspect of NOP does not impose an overhead to the use of network resources.

It is believed that future implementations using the proposed solution will work correctly based on the fact that NOP uses very standardized entities, in the form of factual-executional and logical-causal notifier entities, which comprise now the *Distributed Attribute*.

Acknowledgments

We would like to thank the NOP research group at Federal University of Technology - Paraná (UTFPR), specially Professors Jean M. Simão and Paulo C. Stadzisz who created NOP. We would also like to thank Professor Hervé Panetto from the University of Lorraine

for his review of earlier versions of this work and Alexandre Henzen for the creation of the NOP Java Framework.

References

Agha, G. A. (1985). Actors: A model of concurrent computation in distributed systems. Technical report, MIT Artificial Inteligence Lab.

Alvaro, P., Marczak, W. R., Conway, N., Hellerstein, J. M., Maier, D., and Sears, R. (2011). Dedalus: Datalog in time and space. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 262–281. Springer-Verlag.

Belmonte, D. L., Linhares, R. R., Stadzisz, P. C., and Simão, J. M. (2016). A new method for dynamic balancing of workload and scalability in multicore systems. *IEEE Latin America Transactions*, 14(7):3335–3344.

Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59.

Cook, R. P. (1980). *mod - a language for distributed programming. *IEEE Transactions on Software Engineering*, SE-6(6):563–571.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Pearson, 5th edition.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382.

Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK. Springer-Verlag.

Grumbach, S. and Wang, F. (2010). *Netlog, a Rule-Based Language for Distributed Programming*, pages 88–103. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kendall, S. C., Waldo, J., Wollrath, A., and Wyant, G. (1994). A note on distributed computing. Technical report, Mountain View, CA, USA.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*

Lobo, J., Ma, J., Russo, A., and Le, F. (2012). Declarative distributed computing. *Correct Reasoning*, pages 454–470.

Simão, J. M. (2005). *A Contribution to the Development of a HMS Simulation Tool and Proposition of a Meta-Model for Holonic Control*. PhD thesis, UTFPR.

Simão, J. M., Banaszewski, R. F., Tacla, C. A., and Stadzisz, P. C. (2012). Notification oriented paradigm (nop) and imperative paradigm: A comparative study. *Journal of Software Engineering and Applications (JSEA)*, 5(6):402–416.

Simão, J. M. and Stadzisz, P. C. (2009). Inference based on notifications: A holonic metamodel applied to control issues. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 1:238–250.

Ullman, J. D. (1984). *Principles of database systems*. Galgotia publications.

Xavier, R. D. (2014). Software development paradigms: comparison between events oriented and notification oriented approaches. Master's thesis, UTFPR.