

Fault Tolerance in Heterogeneous Distributed Systems

Zhe Wang, Naftaly H. Minsky
Department of Computer Science
Rutgers University

Email: {zhewang, minsky}@cs.rutgers.edu

Abstract—Dependability of heterogeneous distributed systems is an important issue. Coordination failures may occur even if the given coordination protocol is adhered to by all participants. The *fault tolerance* (FT) properties of systems are difficult to achieve, especially at application level. What is common to current FT-techniques is their reliance on the code of the various system components, which are often required to be written in a specific language. From the viewpoint of distributed systems, such techniques are feasible for homogeneous systems, or at least systems that are designed and maintained by a single administrative domain. But such code-based techniques are generally unreliable for open systems, due to the lack of overall control over the code of components. This leaves open distributed systems vulnerable to their own faults and to attack on them.

However, certain types of FT measures can be established in distributed systems by controlling the flow of messages between system components, independently of the code of system components—which we plan to do via a distributed coordination and control mechanism called Law-Governed Interaction. We demonstrate in this paper, there is a substantial range of FT measures that can be established completely by controlling messaging. Moreover, although the FT-measures to be developed are meant mostly for open systems, some of them can be useful for distributed systems in general, even where traditional code-based techniques are feasible.

I. INTRODUCTION

Heterogeneous distributed systems face two issues: (1) the implementation of system properties, i.e., of properties and regularities that span the entire system, or a substantial subset of its components; and (2) the dependability of such properties, by which we mean the resilience of such properties to failures and attacks.

These issues are particularly problematic in heterogeneous systems whose components may be written in different languages, may run on different platforms, and may be designed, constructed, and even maintained under different administrative domains. Such a distributed system is often said to have an *open architecture*, or just be an *open system*¹ [1], [2]—because of the lack of effective constraints on the organization of the system as a whole, and on the internals of its disparate components. Systems are increasingly designed

¹The term “open system”, as used here, has nothing to do with the concept of open source.

to be open, with the hope that this would make them more flexible. The concept of service oriented architecture [3](SOA) is a prominent example of this trend, which is being adopted by a wide range of complex distributed systems, such as: commercial enterprises, societal and governmental institutions, and various types of virtual organizations.

While the systems of such type are hard to be implemented correctly. The *fault tolerance* (FT) properties of the systems are also difficult to achieve, especially at application level. The need to develop fault tolerance techniques specifically for the application level of systems—sometimes called “*software fault-tolerance*”—has been pointed out already in 1975 by Randell [4], who argued that the traditional FT techniques, designed mostly for hardware failures, are not sufficient for handling the various ways in which an application may fail. This is true, in particular, for coordination failures. Such as a failure of a group of distributed actors to collaborate effectively towards a common goal, or to compete safely over some resources, due to the failure of any one of them to abide by the necessary coordination protocol.

Considerable research effort has been devoted to application level FT—see [5] for a survey. This generally involves incorporating failure-handling code (henceforth “FT-code”) into the software. Various types of FT measures have been developed in this way—such as *exception-handling*, *recovery blocks* [4], and *N-version programming* [5]—for handling various kinds of failures at the application level. The deployment of such techniques suffer from two types of difficulties: (a) they tend to complicate the system, and (b) when a similar, or even same, FT-code has to be used in many system components, their deployment tends to be laborious and error prone. These problems are sometimes alleviated via *meta object protocol* (MOP) [6], which enables what is called *reflection*; and via *aspect oriented programming* (AOP) [7], [8], [9]. Moreover, special programming languages, such as Argus [10], and coherent sets of tools, such as Arjuna [11], were developed for building fault tolerant distributed systems.

What is common to these FT-techniques is their reliance on the code of the various system components, which are often required to be written in a specific language. From the viewpoint of distributed systems, such techniques are

feasible for homogeneous systems, or at least for systems that are designed and maintained under a single administrative domain—which can, for example, ensure that all system components are governed by the same AOP code. But such code-based techniques are generally unreliable for open systems, due to the lack of overall control over the code of the various components, or even of the language in which they are written. *This leaves open distributed systems vulnerable to their own faults, and to attack on them.*

It is our thesis, however, that certain types of FT measures can be established in a distributed systems by controlling the flow of messages between system components, independently of the code of most or of all system components—which we plan to do via a distributed coordination and control mechanism called Law-Governed Interaction (LGI)—an overview of which is provided in Section II. This cannot be done for all FT measures that can be established by inserting suitable code into the components themselves. For example, we cannot ensure orderly checkpointing by selected components—an important basis for many conventional FT-measures. Yet, as we intend to demonstrate in this paper, there is a substantial range of FT measures that can be established either completely by controlling messaging, or with the help of relatively few distinguished actors that can be trusted to carry out the role assigned to them.

Moreover, although the FT-measures to be developed are meant mostly for open systems, some of them can be useful for distributed systems in general, even where traditional code-based techniques are feasible. This for two main reasons: first, our FT-measures would be independent of most of the system code, and cannot be violated by changes in it. Second, enacting such measures would not complicate the code because the mechanism is completely separate from it—this is, in a sense, similar to FT measures implemented via the meta-object protocol, or via AOP, which we can independently of the language in which the components are written.

The rest of this paper is organized as follows. Section II provides a very brief outline of the LGI middleware, which serves as the basis for this work. Section III introduces a generic scheme called Coordinated Atomic Actions that handles coordination failures at application level, and how we implement this scheme via LGI for open systems. Section IV introduces a case study of how to handle the coordination failure during a leadership exchange between police teams. Section V is the implementation of the case study, showing how the behavior of the police teams is regulated via LGI, and how coordination failure is handled. And we conclude in Section VI.

II. THE LAW-GOVERNED INTERACTION (LGI) MIDDLEWARE—AN OVERVIEW

LGI is a middleware that can govern the interaction (via message exchange) between distributed *actors*, by enforcing an explicitly specified law—and possibly multiple laws—about such interaction. We provide here a brief, and rather abstract, overview of LGI; focusing on what is the most relevant to this paper. A more detailed presentation of LGI, and a tutorial of it, can be found in its manual [12]—which describes the release of an experimental implementation of the main parts of LGI. For additional information and examples the reader is referred to a host of published papers, some of which will be cited explicitly in due course.

The rest of this section is organized as follows. We start, with the local nature of the interaction laws under LGI—a key characteristics of this middleware that enables many of the novel features of it. We then discuss the following aspects of LGI: the structure of its laws; and the law enforcement mechanism.

A. The Local Nature of Interaction Laws

Although the purpose of interaction laws is to govern the exchange of messages between different distributed actors, they do not do so directly under LGI. Rather, an LGI law \mathcal{L} governs the interaction of any actor operating under it, essentially by controlling its ability to send messages to others, and to receive messages from them². A law \mathcal{L} is local to each actor x operating under it, in that its rulings are based solely on the local state of x and on the event that occurs at it, and are completely independent of the coincidental state and events occurring anywhere else in the system. Such a law can be enforced locally, and thus very scalably, in a manner described in Section II-C. Moreover, the locality of LGI laws has several other beneficial consequences, some of which will be pointed out in due course.

It should also be pointed out that although locality constitutes a strict constraint on the structure of laws, it does not reduce their expressive power. This has been proved in [12]. In particular, despite its *structural locality*, an LGI law can have *global effect* over what is called an \mathcal{L} -community, defined as the set of actors operating under a common law \mathcal{L} .

B. LGI Laws—a Definition

An *interaction law* (or simply a *law*) \mathcal{L} is defined over three elements—described with respect to a given actor x that operates under this law: (1) a set E of *interactive events* that may occur at any actor, including the arrival of a message at x , and the sending of a message by it; (2) the *state* (also called the *control-state*) S_x associated with each

²In fact, a law can also cause messages to be changed and rerouted, and it can change the state of an agent.

actor x , which is distinct from the internal state of x , that is invisible to the law; and (3) a set O of *interactive operations* that can be mandated by a law, to be carried out at x upon the occurrence of interactive events at it; this set includes operations that forward messages to others, along with some other types of operations that have an effect on the flow of message into x and from it.

Now, the role of a law under LGI is to decide what should be done in response to the occurrence of any interactive event at an actor operating under this law. This decision, with respect to actor x , is defined by the following mapping:

$$\mathcal{L} : E \times S_x \rightarrow S_x \times (O)^* \quad (1)$$

In other words, for any given $(event, state)$ pair, the law mandates a new state (which may imply no state change), as well as a (possibly empty) sequence of interactive operations. Note, in particular, that the ruling of the law at a given moment of time depends on the state of x at that moment; and that the evolution of the state itself is determined by the law, and by the history of interactive-events at x . LGI laws are, therefore, stateful, and sensitive to the history of interaction.

Note that the law is a complete function, so that any mapping defined by Formula 1 is considered a valid law—which means that a law of this form is *inherently self consistent*. This does not mean, of course, that a law cannot be wrong. It can be wrong in the sense that it does not work as intended by its designer; but this is not a matter of inconsistency.

Finally, it is worth pointing out that while Formula 1 is a definition of the semantics of laws³, it does not specify a language for writing laws. In fact, the current implementation of LGI supports two different *law-languages*, one based on the logic-programming language Prolog, and the other based on Java. But the choice of language has no effect on the semantics of LGI, as long as the chosen language is sufficiently powerful to specify all possible mappings defined by Formula 1.

C. The Decentralized Law Enforcement Mechanism

Consider an actor x that chooses to operate under a law \mathcal{L} . It can do so by *adopting* a generic controller as its mediator, loading law \mathcal{L} into it. Once thus adopted, this controller is denoted by $T_x^{\mathcal{L}}$ —meaning that it operates under law \mathcal{L} , serving actor x —and the pair $\langle x, T_x^{\mathcal{L}} \rangle$, is called *agent x* and is referred to as an *\mathcal{L} -agent*—and sometimes simply an “agent”. This adoption, which signifies the *birth* of agent x , is one of the interactive events of LGI, so that the law in question has the possibility of refusing to be adopted by this actor,

³Modulo the fact that the sets E of events and O of operations have not been fully spelled out here.

and can mandate some initialization for it, if it does not refuse.

Note the fundamental difference between a bare actor and its agent: while the interactive behavior of an actor is unpredictable—unless its code is known—the interactive behavior of an \mathcal{L} -agent is known to conform to law \mathcal{L} .

Figure 1 depicts the manner in which a pair of agents, operating under possibly different laws, exchange a message (An agent is depicted here by a dashed oval that includes an actor and its controller). Note the *dual nature* of control exhibited here: the transfer of a message is first mediated by the sender’s controller, subject to the sender’s law, and then by the controller of the receiver, subject to its law. This dual control, which is a direct consequence of the local nature of LGI laws, has some important consequences. In particular, it facilitates flexible interoperation and it enables more sophisticated control than possible under many AC mechanisms that provide control only on the receiver side.

The overhead incurred by this kind of control turns out to be relatively small. In circa 2000 it was measured to be around 50 microseconds for fairly common laws, which is negligible for communication over WAN. This is one of the results of a comprehensive study of this overhead in [13].

Finally, we note that a generic controller needs to be trusted to enforce correctly any law it is adopted with. There are several ways for providing such trusted controllers as the TCB (Trusted Computing Base) of the system in question. In the case of a bound OSN, like our B_E example, we expect this to be done by the enterprise E , in the context of which B_E operates. This company could construct what is called a *controller service* (CoS) that maintains a set of well tested *controller pools*, each of which can host a number—it is usually in the hundreds—of individual controllers that can be used by arbitrary actors, upon request. For other types of OSNs one expects the CoS to be maintained by some commercial company that provides its services for a fee.

Note, therefore, that a controller $T_x^{\mathcal{L}}$ and the actor x that adopted it would run on different hosts. This would help prevent x corrupting its own controller. Even if a controller is hacked, since it does not keep the messages it passes, there is no way to get the information of the whole history. And since it would be much harder to compromise many controllers than one, the global view of the whole system will not be obtained.

III. COORDINATED ATOMIC ACTION IN OPEN SYSTEMS

Coordination failures may occur even if the given coordination protocol is adhered to by all its participants. For example a leader election protocol may not ensure that a leader got a majority vote. Another type of failure may be caused by lost messages. Many such failures can be handled via the concept *Coordinated Atomic Actions* (CAAction),

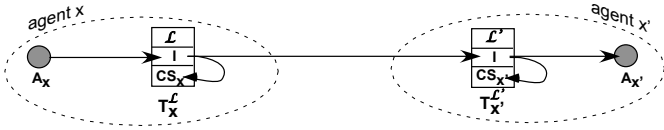


Fig. 1. Interaction between a pair of agents, mediated by a pair of controllers under possibly different laws.

introduced by Randell et al. [14]. This is a generic, broad spectrum, and quite influential [15] scheme, which can be described, broadly, as follows: A CAAction is a kind of virtual box—which needs to be tailor-made for any given type coordination activity. To engage in a given coordination activity, the participants “enter” a suitable CAAction, and then must operate subject to its control. The host CAAction ensures the ACID property for this activity, and provides either forward or backward recovery, if the activity fails. We mention here two key elements of this scheme: (a) a set of constraints on the behavior of the participants, before they enter a CAAction⁴, and once they are in it; (b) a supervisory code that handles the recovery from failures, and other matters.

Now, the concept of CAAction has been implemented in a central systems, and in monolithic distributed systems [16]—where one can ensure that all participants entering a given CAAction adhere to its constraints, by programming them accordingly. But since such code-based implementations are not dependable in open systems, to make CA Actions feasible in open systems, there are two more properties we need to provide: 1) operation capability without knowing the implementation detail of each component; 2) the enforcement of the components’ behavior on fault tolerance and exception handling. We implement this scheme via LGI, roughly as follows: (1) a given type of CAAction will be defined by a law that enables its participants to engage in the given coordination activity, while imposing the required constraints on their behavior in the CAAction and out of it (see (a) above); and (2) the required supervisory code (see (b) above) will be programmed, in the conventional sense, in some trusted actor. It should be pointed out, however, that we will not be able to implement all aspects of the original CAAction, for all types of coordination activities. In particular, we will not be able to provide backward recovery if it requires checkpointing, because LGI cannot ensure checkpoints by individual actors.

IV. LEADERSHIP EXCHANGE—A CASE STUDY

Consider three teams (T1, T2 and T3) of traffic police officers, each of which handles the traffic of a certain region by operating a set of traffic-related devices, such as draw bridges, traffic lights, or road blocks. They can do this via a collection of sensors and actuators distributed in their region,

⁴In particular, they should not engage in the given activity outside of the CAAction

which constitutes an open system. Moreover, each team is managed by a commander (C1, C2 and C3), who assigns the team members to various tasks, monitors their progress, and exerts control over what each team member can do. The command that the commander sends to team member could be about querying the sensors on the road, operating on those actuators, or granting the team member access control rights to devices. The members of a team will only obey the command from its commander. And the commander is able to execute the power of its role by presenting the baton of the role.

Suppose there is a need for three commanders to switch their roles. They can discuss with each other and delegate its commander role by sending the baton to the successor. However, sending the role baton is asynchronous for each commander in open systems and the execution of the exchange agreement is not guaranteed, there are two serious situations could occur: 1) before a commander C1 send its role Commander(T1) and baton to the other one C2, it duplicates its baton and only sends the copy to C2. After that it continues claiming it is the commander of that team Commander(T1). Then there will be two commanders for T1 and the team will be confused about two command sources and possible contrary command contents; 2) What’s even worse is that if a commander C1 receives role Commander(T2) from commander C2, it can refuse to send role Commander(T1). As a result, it will have the commander role of two teams, which is a very dangerous situation for obvious reason.

To avoid those situations, we employ CA Actions to execute the role exchange. The essence of CA actions is that for the outside world(three teams), the activity and communication between involved actors is atomic. To achieve this, our system has following according properties:

- 1) Entering, leaving and operating inside the action will be managed by actor in a supervisor role: The commanders may get in to the action freely. But they can not leave the CA action until they have an agreement on the activity result. Two commanders have same role or one commander has two roles can not pass the acceptance test.
- 2) The actors might lose some power while inside a CA action: The commanders can not send command to anybody when inside the CA Action. However, if the commanders receive report from its team members during the exchange, it can hold it and process it or forward it afterward.
- 3) The actors may also gain some power while inside a CA action: The commanders can only send role to each other when inside the CA Action.

V. IMPLEMENTATION OF LEADERSHIP EXCHANGE

The law $\mathcal{L}_{\mathcal{P}}$ is used for regulating every aspect of the operations and behaviors of the systems. We split it into several parts according to their functionalities. In Section V-A,

we show how commanders talk to its team and to each other. In Section V-B, we elaborate how the commanders get into a CAAction and how to get out of a CAAction if they all pass the acceptance test or some failure happens. In Section V-C, we demonstrate how the commanders exchange their leadership while inside a CAAction. And we show an exception handling is triggered in Section V-D.

A. Communication Outside of CAAction

There are three roles in this police systems—regular police officer who belongs to a team, commander of a team and a supervisor who takes care of role exchange. They communicate with each other by sending messages using their electronic devices. A message must have a header and/or a body. Some message headers can be used by anyone, some cannot. For example, “freeTalk” header is used by anyone for conversing with another officer, while “command” header can only be issued by a commander to its team. For the sake of simplicity, we assume that all the messages will arrive at their destination within a certain amount of time. As a police officer, it has a “team(Ti)” in its control state by providing its baton when entering this system. And a commander has a “commander(Ti)” in its control state by the same way of providing its baton for its commander role. Rule $\mathcal{R}1$ shows that a commander of a team can send command only to its team members, while it is not in the stage of role exchange. If the commander is in the stage of role exchange, it will have “state(CAA)” in its control state. We will explain this in one of following rules. When a team member receives a command from its commander, as in Rule $\mathcal{R}2$, it will execute it immediately. As has been demonstrated in [17], the command could be querying the sensors on the road, operating on those actuators, or granting the team member access control rights to devices. And the officer will execute it accordingly. Whether it is the member of team i is determined by whether it has “team(Ti)” in its control state, which is added when it joins the system and provides its baton for its team. The team member will not need to concern about whether the command is really from the commander, because that is guaranteed by Rule $\mathcal{R}1$. Rule $\mathcal{R}3$ shows that a person (no matter it is commander, team member or supervisor) can talk to others freely as long as it is not using the keywords, such as “command”, “delegate” or “exchange”. The free talk between two persons is not restricted by its team or role. Thus, it is very useful for commanders to communicate for initiating a role exchange or distributing roles. And Rule $\mathcal{R}4$ shows that a person can receive and see the free talk message from anyone.

B. Entering and Leaving CAAction

When there is a need for exchange roles, a commander can request to exchange roles by sending the request, individually or jointly, to a supervisor, as in Rule $\mathcal{R}5$. After the supervisor receives a role exchange request from a commander, it will wait a certain time for exchange request from other

$\mathcal{R}1$ UPON sent (Commander, command (Ti, command), X) if (Commander (Ti)@CS and \neg state (CAA)@CS) do (Forward)
$\mathcal{R}2$ UPON arrived (Commander, command (Ti, command), X) if (team (Ti)@CS) execute (command)
$\mathcal{R}3$ UPON sent (X, freeTalk (M), Y) do (Forward)
$\mathcal{R}4$ UPON arrived (X, freeTalk (M), Y) do (Deliver)

Fig. 2. Communication Outside of CAAction Part of the Law \mathcal{L}_P

commanders. If there are more requests, it will notify those commanders to get into the exchange stage and exchange their roles. If there is no other request, it will ignore the request back to that commander. After the supervisor notifies the commanders to exchange their roles, it will wait for their confirmations and impose an obligation for recovery in case the exchange doesn’t succeed. During this period of time, the commanders can talk freely as we described in Rule $\mathcal{R}3$ and delegate its role to each other, which we will discuss in next section. Rule $\mathcal{R}8$ shows that after receiving a message of getting into the stage of role from the supervisor, the commander will be added a control state “state(CAA)”, under which the commander will lose a certain power, like issuing a command. It will also keep the record of its role history for the purpose of recovery. When the exchange is done, all the commanders will send the supervisor a confirmation message. After the supervisor receives the confirmations from all the commanders, which means the role exchange succeeded, it will notify them to get out of the exchange stage and repeal the obligation for recovery through Rule $\mathcal{R}9$. When a commander receives a notification that the exchange is finished, it will remove the “state(CAA)” control state and exercise its commander role as shown in Rule $\mathcal{R}10$. The backup for the exchange will be collected as garbage.

C. Leadership Exchange

During the exchange stage, the commanders can not issue a command to its team members. It can only discuss with the other commander, delegate its role or reject the exchange. The commanders cannot get out of the CAAction until everyone receives the notification from the supervisor. We don’t regulate which commander should send its role to whom. It’s up to them to make the delegation decision. If a commander decides to send its role to another commander, as in Rule $\mathcal{R}11$, it has to relieve that role. This is enforced

```

 $\mathcal{R}5)$ 
  UPON sent (CommanderX, requestEnter,
    Supervisor)
  if (Commander (Ti) @CS)
  do (Forward)

 $\mathcal{R}6)$ 
  UPON arrived (CommanderX,
    requestEnter, Supervisor)
  do (Add (CommanderX, requestList))
  do (Deliver)

 $\mathcal{R}7)$ 
  UPON sent (Supervisor,
    admit (requestList), requestList)
  imposeObligation (recover, timeout)
  do (Forward)

 $\mathcal{R}8)$ 
  UPON arrived (Supervisor,
    admit (requestList)), CommanderX)
  do (+ state (CAA) @CS)
  do (+ former (Commander (Ti)) @CS)
  do (Deliver)

 $\mathcal{R}9)$ 
  UPON
    sent (Supervisor, exit, requestList)
    repealObligation (recover)
  do (Forward)

 $\mathcal{R}10)$ 
  UPON arrived (Supervisor, exit,
    CommanderX)
  if (state (CAA) @CS and
    former (Commander (Ti)) @CS)
  do (- state (CAA) @CS)
  do (- former (Commander (Ti)) @CS)
  do (Deliver)

```

Fig. 3. Entering and Leaving CAAction Part of the Law \mathcal{L}_P

to avoid the situation of two commanders for one team. Moreover, it can only send its role when it is inside the exchange stage. Rule $\mathcal{R}12$ shows that a commander can become the leader of a team only when it is in the stage of exchange and receives the role from another commander. It can not issue command to its new team member since it is still in the exchange state. Furthermore, it will send a copy to the supervisor showing it receives the role. According to Rule $\mathcal{R}13$, a commander can send the confirmation message to the supervisor, if there is one commander role under its control state. The supervisor will send out the notification after it receives the confirmation from every commander.

D. Exception Handling

There are several types of possible exceptions could happen in such system. One is during the exchange stage, one or more commanders' devices is disconnected from the systems. In this case, it may haven't sent its role or received from the

```

 $\mathcal{R}11)$ 
  UPON sent (CommanderX,
    delegate (Commander (Ti)),
    CommanderY)
  if (state (CAA) @CS)
  do (- Commander (Ti) @CS)
  do (Forward)

 $\mathcal{R}12)$ 
  UPON arrived (CommanderX,
    delegate (Commander (Ti)),
    CommanderY)
  if (state (CAA) @CS)
  do (+ Commander (Ti) @CS)
  do (Deliver)

 $\mathcal{R}13)$ 
  UPON sent (CommanderX, confirm,
    Supervisor)
  if (Commander (Ti) @CS )
  do (Forward)

 $\mathcal{R}14)$ 
  UPON arrived (CommanderX, confirm,
    Supervisor)
  do (Deliver)

```

Fig. 4. Leadership Exchange Part of the Law \mathcal{L}_P

other, or at least haven't sent out the confirmation message to supervisor. After the supervisor notifies the commanders to exchange their roles, it will wait for their confirmations. If it doesn't receive all the confirmations within a certain amount of time, it will consider there is an exception in the exchange, like in Rule $\mathcal{R}15$. Due to its recovery policy, it can use backward recovery (all commanders go back to their former teams), forward recovery (the supervisor designates the appointments) or combined recovery to handle that exception. The according exception handling policy will be applied by the supervisor. If the supervisor choose to use backward recovery, it will inform all the commanders to restore its former duty. Rule $\mathcal{R}16$ shows that after the commander receives the exchange recovery message from the supervisor, it will resume its former role based on the backup and get out of the exchange stage.

VI. CONCLUSION

This paper introduces a fault tolerance mechanism of handling coordination failures for heterogeneous distributed systems. Common FT-techniques at application level require code-injection, which is not achievable for open systems, since the lack of control over the code of the components.

We propose a coordination mechanism called Law-Governed Interaction (LGI) to control the flow of messages between system components, independent from the code of system components. By providing a range of FT measures that can be established by controlling messaging, the

```

 $\mathcal{R}15$ )
UPON obligationDue(recover)
do(Forward(Supervisor, recover,
requestList))

 $\mathcal{R}16$ )
UPON arrived(Supervisor, recover,
CommanderX)
former(Commander(Ti))@CS
do(+ Commander(Ti))@CS
do(- state(CAA))@CS
do(- former(Commander(Ti))@CS)
do(Deliver)

```

Fig. 5. Exception Handling Part of the Law \mathcal{L}_P

coordination failures within an open system can be handled. This mechanism for fault tolerance can be used for distributed systems in general.

This mechanism is implemented for case study of leadership exchange between police teams. The preliminary testing and experiments of our implementation show that our method is feasible and promising.

REFERENCES

- [1] C. Bidan and V. Issarny, "Dealing with multi-policy security in large open distributed systems," in *Proceedings of 5th European Symposium on Research in Computer Security*, Sep. 1998, pp. 51–66.
- [2] A. Artikis, M. Sergot, and J. Pitt, "Specifying norm-governed computational societies," Department of Computing, Imperial College of Science Technology and Medicine, London, Tech. Rep., 2006.
- [3] M. P. PAPAZOGLU, P. TRAVERSO, S. DUSTDAR, and F. LEY-MANN, "Service-oriented computing: A research roadmap," *International Journal of Cooperative Information Systems*, vol. 17, no. 2, pp. 223–255, 2008.
- [4] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, no. 6, pp. 437–449, Apr. 1975. [Online]. Available: <http://doi.acm.org/10.1145/390016.808467>
- [5] V. D. Florio and C. Blondia, "A survey of linguistic structures for application-level fault tolerance," *ACM Comput. Surv.*, vol. 40, no. 2, 2008. [Online]. Available: <http://dblp.uni-trier.de/db/journals/csur/csur40.html#FlorioB08>
- [6] G. Kiczales, J. Rivieres, and D. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with aspectj," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, October 2001.
- [8] A. Zarras, M. Fredj, N. Georgantas, and V. Issarny, "Engineering reconfigurable distributed software systems: Issues arising for pervasive computing," in *Fault-Tolerant Systems*, ser. LNCS, M. Butler et al., Eds. Springer Verlag, 2006, vol. 4157, pp. 364–386.
- [9] M. Caporuscio, A. D. Marco, and P. Inverardi, "Model-based system reconfiguration for dynamic performance management," *Journal of Systems and Software*, vol. 80, no. 4, pp. 455 – 473, 2007, software Performance 5th International Workshop on Software and Performance. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121206002068>
- [10] B. Liskov, "Distributed programming in argus," *Commun. ACM*, vol. 31, no. 3, pp. 300–312, Mar. 1988. [Online]. Available: <http://doi.acm.org/10.1145/42392.42399>
- [11] S. K. Shrivastava, "Lessons learned from building and using the arjuna distributed programming system," in *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*. London, UK, UK: Springer-Verlag, 1995, pp. 17–32. [Online]. Available: <http://dl.acm.org/citation.cfm?id=647369.723769>
- [12] N. H. Minsky, *Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism (An Introduction, and a Reference Manual)*, February 2006, (available at <http://www.moses.rutgers.edu/>).
- [13] N. H. Minsky and V. Ungureanu, "Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems," *TOSEM, ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 3, pp. 273–305, July 2000.
- [14] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery," in *FTCS*. IEEE Computer Society, 1995, pp. 499–508. [Online]. Available: <http://dblp.uni-trier.de/db/conf/ftcs/ftcs95.html#XuRRRSW95>
- [15] D. P. Pereira and A. C. V. de Melo, "Formalization of an architectural model for exception handling coordination based on ca action concepts," *Sci. Comput. Program.*, vol. 75, no. 5, pp. 333–349, May 2010. [Online]. Available: <http://dx.doi.org/10.1016/j.scico.2009.12.006>
- [16] J. Xu, A. B. Romanovsky, and B. Randell, "Coordinated exception handling in distributed object systems: From model to system implementation," in *ICDCS*, 1998, pp. 12–21. [Online]. Available: <http://dblp.uni-trier.de/db/conf/icdcs/icdcs98.html#XuRR98>
- [17] R. Dudheria, W. Trappe, and N. Minsky, "Coordination and control in mobile ubiquitous computing applications using law governed interaction," in *Proc. of the Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM) Florence, Italy*, October 2010, pp. 247–256.