ISR
Institute for Software Research
University of California, Irvine

Accountability Through Architecture for Decentralized Systems: A Preliminary Assessment

Matias Giorgio
University of California, Irvine
mgiorgio@uci.edu

Richard N. Taylor
University of California, Irvine
taylor@uci.edu

October 2015
ISR Technical Report # UCI-ISR-15-2

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
isr.uci.edu

Accountability Through Architecture for Decentralized Systems: A Preliminary Assessment

Matías Giorgio, Richard N. Taylor

Institute for Software Research

University of California,
Irvine

Technical Report
UCI-ISR-15-2

October 2015

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Decentralized systems, that is, distributed systems designed, developed, operated and maintained by more than one authority affect people's lives every day. Representative domains where decentralized systems operate include e-commerce, healthcare, inter-government coordination, emergency response, and electronic trading. Such systems present unique challenges in terms of evolution, adaptation, and security. It is difficult, if not impossible, to coordinate the evolution of a decentralized system as organizations evolve the system's constituent components in response to their potentially independent organizational needs and interests. Security is a major concern since there is no single, uniform perimeter to defend, and it is significantly affected by complex trust relationships, susceptible to change at any point of time; a trusted component can become the epicenter of an "insider attack" if someone takes control over it. Worse still, decentralized systems are the supreme example of systems of systems, therefore, an unintentional mistake or an unexpected error can put at risk the system's integrity and the services offered. At the core of this research study is the concept of capability, an unforgeable reference whose possession confers both the right and authority to perform some action within a system. We hypothesize that capability accounting – tracking the creation, exploitation, and transfer of capabilities – let us obtain insightful information about a system, therefore, help us build, operate and maintain secure decentralized systems. We ground our work in COmputationAl State Transfer (COAST), an architectural style for secure and adaptive decentralized systems that permits and encourages continuous auditing. In COAST, capabilities are first-class architectural elements that regulate and articulate what a computation may do, and when, how, and with whom a computation may communicate. This work presents an assessment of capability accounting within the financial trading domain. It includes a framework that specifies which capability events are to be studied, and the means to represent, capture and examine those events as well as techniques to analyze them. We evaluate the proposed framework and techniques

using COast Monitoring Event Tool (COMET), a tool we built for analyzing capabilities, a prototype of an electronic trading system, and various trading computations. We found that capability accounting is a valuable technique to obtain information about a system, and that COAST is very well-suited for this form of measurement.

Chapter 1

Introduction

A decentralized system is a distributed system for which there is no central administrative authority to dictate how the distributed subsystems must be developed, operated, and maintained [21]. In decentralized systems service providers and service clients can each operate under different authority and evolve independently. Examples of such systems include e-commerce, healthcare, inter-government coordination, and electronic trading.

In decentralized systems, security is deeply affected by complex trust relationships among participant systems that can change at any point of time. If a system is breached, all the information it accesses may be compromised, and all the data it produces should no longer be trusted. Ideally, a breached system should be isolated, and all interactions that include it should be reorganized in order to minimize the impact. Service provisioning will be potentially affected at many levels, from the services offered by the compromised system, processes in which that system participates, to even the functionality expected from the entire decentralized system. Such systems are not immune to "insider attacks", one component can abuse a right legitimately granted at a previous time. Furthermore, even in a world of perfect security, verifying the correct operation and integrity of critical software systems is paramount in order to ensure that services are offered as providers and consumers expect.

Designing and implementing these systems is a substantial challenge. How can one ensure

service, adaptivity, and security when individual subsystems are evolving independently with little or no coordination?

We propose to address these questions from the perspective of software architecture [39] and architectural styles [45]. Valued software properties arise from software architecture: scalability, evolvability, dynamism, traceability [32], performance, reliability, and cost [30, 40], only to name a few. Software architecture can also arise *accountability*, that is, the degree to which it can justify actions, effects, qualities, and properties associated to the system it grounds, and facilitate the obtention of insightful information about how the system behaves.

Among the many forms of measurement that can be used to evaluate and understand systems behavior, this research study focuses on *capability accounting*, in which the principal unit of analysis is the concept of *capability* (an unforgeable right to perform some action within a system [10]). We propose that capability accounting – tracking the creation, transfer, and exercise of capability – is fundamental to the debugging, verification, and forensic auditing of decentralized systems.

This work is grounded on **COmputationAl State Transfer (COAST)** [21], an architectural style for secure and adaptive decentralized systems. COAST constructions permit and encourage continuous accounting and systemic auditing to verify the correct operation and integrity of critical elements of a decentralized system. COAST provides new principles to support the design of openly secure decentralized applications, and addresses security as a fundamental concern. COAST targets decentralized applications where organizations offer services formulated as execution hosts (called peers) and third-party organizations create their own custom-tailored versions of services by dispatching computations to asset-bearing peers. Decentralized security and guarding against untrusted or malicious mobile code are principal concerns. To this end, COAST offers two distinct forms of capability, (1) **functional capability**: what a computation may do, and (2) **communication capability**: when, how, and with whom a computation may communicate. This work focuses on communication capability accounting, that is, the activities involved in acquiring, examining and

acting using capabilities as the principal unit of analysis.

In COAST, communications between peers are both granted and constrained by **capability URLs (CURLs)**. A CURL conveys the ability for two computations to communicate. A CURL c issued by a computation x, is an unguessable, unforgeable and tamper-proof reference to x that grants to any computation y holding c the power to transmit messages to x.

The reference COAST implementation used throughout this study is Motile/Island [22], a new generation of infrastructure that supports more sophisticated forms of computational exchange.

1.1 Research

1.1.1 Research Goals

This research work explores capability accounting in COAST systems, and focuses on communication capabilities. The research goals are:

- Assess feasibility of capability accounting to help design, develop, operate, and maintain COAST systems.
- Propose a framework to structure the practice of capability accounting.
- Explore means to structure and analyze capability events to obtain insightful information about a system.

Assessing feasibility of capability accounting

As part of this research work, the COAST reference implementation (Motile/Island platform) was augmented in order to capture communication capability events and store them in a database for inspection and analysis as shown in Figure 1.1. Additionally, A tool called

COMET (COast Monitoring Event Tool) was built to inspect and analyze capability events, detect event patterns, highlight useful information, and report anomalies.

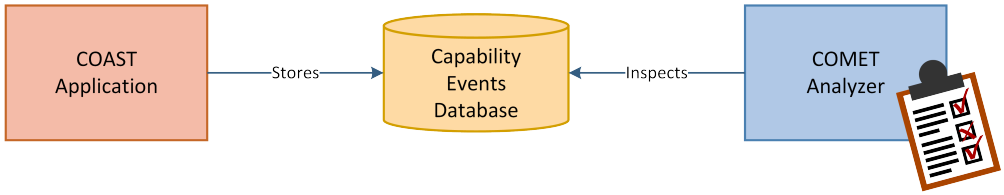

Figure 1.1: COAST applications capture capability events and store them in a database. COMET performs different types of analyses using the collected data.

COMET performs evaluation-based analysis, where observed event patterns and expected patterns are compared, in order to determine whether a system behaves as it should.

For the evaluation of this study, a COAST-based prototype of an electronic trading system was developed to illustrate how capability accounting can be used by developers to: debug a system, inspect the behavior of a component, and monitor its security. The prototype models five of the most important components involved in trading: a Market Data Server, a Risk Management Server, an Order Router, an Execution Host, and a Trading Computation[1]. It also represents the interaction of three distinct organizations: a Trading Firm, a Broker, and an Exchange.

It is known in advance that any trading computation must perform some actions and interactions with the components offered by the Broker and the Exchange. COMET was used to analyze capability events produced by the execution of the prototype, and verify its proper execution, that is, to show evidence that the expected actions and interactions were carried out, and that no potentially malicious actions have taken place.

A framework for capability accounting

The framework for capability accounting proposed in this study encompasses a definition of which capability events are to be captured and examined, and a comprehensive, end-to-end

[1]The prototype design was inspired by the Argo Trading Platform: http://www.argocons.com/platform.html.

list of activities to guide the implementation of capability accounting.

Three different types of events are captured and analyzed: the creation, exploitation, and transfer of communication capabilities. **Creation** occurs when a COAST peer creates a CURL which allows other peers to send messages to it. **Exploitation** occurs when one peer sends a message to another peer via a CURL. **Transfer** occurs when a peer sends a CURL embedded in a message to another peer.

The framework includes a series of activities that articulate the means to acquire and examine capability events in compliance with technical, organizational, business-related, and legal protocols and regulations, and act upon the evaluation results of capabilities analysis. The activities included in the model are: **capture**, **transfer**, **storage**, **query**, **preparation**, **evaluation**, **reporting**, and **action**.

Representation of capabilities and analysis techniques

A number of evaluation techniques and operators are proposed by this research study, and implemented in COMET. They were used as part of the evaluation to examine the capability events generated by the execution of the prototype and the trading computations.

1.1.2 Research Questions

The research questions I embarked to answer in this research study are the following:

- Is capability accounting a well-suited form of architectural accountability?
- How can capability accounting be used to help build, operate and maintain secure decentralized systems?
- Is the financial trading domain a suitable domain for capability accounting?

1.2 Organization of this work

The organization of this work is as follows. This chapter presents the topic of decentralized systems, and poses their salient issues. It explains the focus of this study – architectural accountability and capability accounting – introduces COAST, the architectural style used for this work, and describes the research goals and research questions I proposed to answer. Chapter 2 revisits the COAST architectural style. It explains its principles, structural elements and concepts, and rationale, as well as provides an overview of Motile/Island, the reference implementation. Chapter 3 presents capability accounting, the elements that are to be studied, an end-to-end framework for capability accounting implementation, and two distinct analysis: evaluation-based and exploratory. Chapter 4 describes the implementation followed by this work, the tools and technological stack used, and precisely defines how the activities in the proposed framework are carried out to implement capability accounting. Chapter 5 describes COMET, the main tool used in this study to examine capability events. It explains its high-level architecture, principal components, and techniques and operators it offers to analyze capabilities. Chapter 6 provides an overview of the electronic trading domain, why it is a relevant domain for decentralized systems, its impact on the global economy, and major software-related incidents occurred in the last five years as well as the challenges on designing, developing, operating, and maintaining decentralized systems within the domain. Chapter 7 presents the evaluation of this work. A prototype of an electronic trading system, various computations, and all the required infrastructure to support capability accounting were developed in order to explore the topic of this research work and to validate the findings. Chapter 8 discusses lessons learned, reflections, and issues that arose throughout this work. In particular, it focuses on the benefits of applying capability accounting from application-level development, considerations that must be taken depending on the kind of analysis that is to be performed, and capability accounting on domain-dependent and domain-independent issues. It finally presents a real case on financial trading based on a technique called spoofing, and proposes an approach based on COAST and capability

accounting to mitigate the encountered consequences. Chapter 9 concludes this work revisiting what has been done, my findings, questions that still remain open, and suggestions for future work.

Chapter 2

The COAST architectural style

An architectural style is designed to produce architectures that arise certain properties and qualities. Software architecture is the set of principal design decisions about a system, governs the most essential aspects of how a software system is designed, and elicits beneficial qualities in the resulting systems [45].

The **COmputAtional State Transfer (COAST)** architectural style plays a fundamental role in this study of capability accounting. There are four essential reasons that explain why COAST is a natural fit for this work, whose context is secure decentralized systems and Service-Oriented Architectures (SOA):

1. COAST implements capability-based security. To this end, capabilities are well-defined abstractions, treated as first-class architectural elements, making COAST *accountability-friendly*, and a natural candidate to evaluate capability accounting.

2. COAST is specifically designed for decentralized systems: all the constraints imposed by COAST are intended to help developing secure decentralized systems with strong emphasis in adaptability.

3. In COAST, security is enforced from the ground up; it is an aspect present in all decisions that designers and developers make.

4. COAST offers fundamental support for SOA systems via computational exchange and mobile code.

The rest of this chapter explains the fundamental principles and structural elements of COAST, and introduces the Motile/Island reference implementation.

2.1 Principles

COAST is based on the following principles [21]:

- All services are computations whose sole means of interaction is the asynchronous messaging of closures (functions plus their lexical-scope bindings), continuations (snapshots of execution state), and binding environments (maps of name/value pairs).

- All computations execute within the confines of some execution site (E, B) where E is an execution engine and B a binding environment.

- All computations are named by **Capability URLs (CURLs)**, unforgeable, cryptographic structures that convey the authority to communicate. Therefore, computation x may deliver a message (closure, continuation, or binding environment) to computation y if and only if x holds a CURL c_y of y. The interpretation of a message delivered to computation y via CURL c_y is c_y-dependent.

Figure 2.1 illustrates computational exchange within a COAST environment. Peer P_1 is zoomed-in to display the internal components of a COAST peer. It contains four computations x_1, x_2, x_3, and x_4, two Execution Engines, and three Binding Environments. Peer P_2 runs computation x_5, who holds CURLs c_1 and c_3. Peer P_3 runs one computation x_6, who holds a CURL c_3.

The computation x_5 holds CURLs c_1 and c_3, issued by computations x_1 and x_3 respectively, and computation x_6 holds c_3. Since CURLs are the only means to communicate, the only valid communications in this scenario are from x_5 to x_1 via c_1, x_5 to x_3 via c_3, and x_6

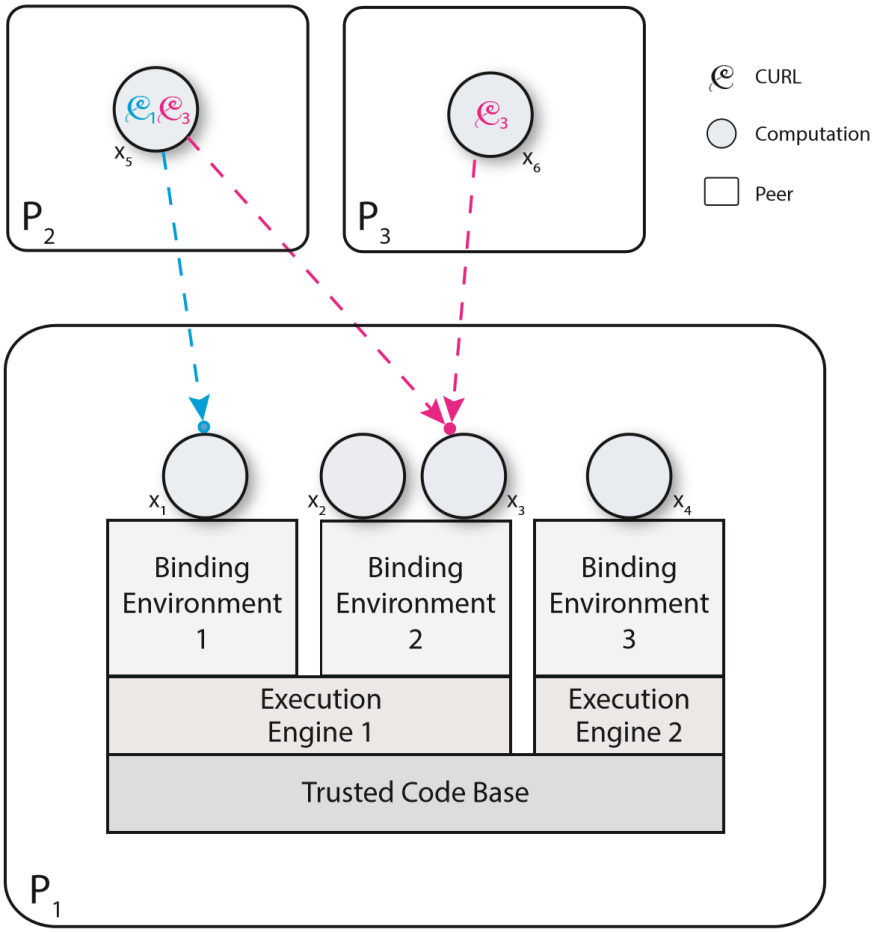

Figure 2.1: Three peers interacting following the COAST style. Peer P_1 is zoomed-in to illustrate COAST architectural elements.

to x_3 via c_3. It is worth noting that communications are unidirectional, that is, if x has the right to send a message to y, it does not imply that y can send a message to x. For that to happen, y needs to hold a CURL issued by x.

COAST relies on two security principles: the Principle of Least Authority (POLA) [41] and capability-based security [10]. POLA dictates the methodical and deliberate allocation of minimal capability across the entire system; that is, the default response upon an accessibility question is to deny, unless the principal can prove that authority should be granted. In capability-based security, a capability is an unforgeable key that grants to its holder both rights and authority. In COAST, the right to send a message to a computation x is granted by holding a CURL issued by x.

2.2 Functional and communication capabilities

A capability [16] is an unforgeable reference whose possession confers both authority and rights to a principal. COAST differentiates between functional and communication capabilities. The former refers to the functions available to a computation (i.e. its binding environment). The latter conveys to computations the ability to transmit messages to other computations or receive messages from other computations.

2.3 Architectural elements

2.3.1 Execution sites

An execution site (E, B) is defined as a pair *(execution engine, binding environment)* and works as a host where computations are confined to run. Multiple execution environments can exist within the same host. An execution engine defines the semantics in which a computation is evaluated, and can change from one computation to another. Examples of execution engines are a Java Virtual Machine, a Racket interpreter and a Javascript JIT compiler. A binding environment is a key/value map that represents all the functions and global variables available to a computation. All the free variables of a computation must be resolved within the binding environment or the execution of the computation is terminated. A binding environment is the architectural element that reifies functional capabilities in COAST, it determines what a computation can (and cannot) do.

2.3.2 Capability URLs (CURLs)

CURLs convey the ability to communicate between computations. A CURL u issued by a computation x is an unguessable, unforgeable, tamper-proof reference to x, as it contains cryptographic material identifying x and is signed by x's underlying execution host [21]. A CURL issued by, and referencing x may be held by one or more computations y. The

CURL u designates the address of computation x, contains arbitrary x-specific metadata (including closures), and enables any computation y holding u to transmit messages to x. When a message is sent using a CURL, both the message and the CURL are delivered to the recipient. Computations use the CURLs they issue to constrain their interactions with other computations and to bound the services they offer. Three main benefits arise from the use of CURLs:

- **On-demand custom-tailored services:** CURLs denote the services offered by a computation x to computations y holding a CURL issued by x. As a result, a computation x can offer different implementations to clients by issuing different CURLs. The evaluation of a message m that arrives via CURL c is constrained and governed by CURL c's execution environment as it defines the set of functions and global variables available to the message m.

- **Powerful communication model:** Computation y can send a message to another computation x, if and only if, y holds a CURL c issued by x. Consequently, interactions among computations is restricted by the possession of their CURLs. Additionally, a CURL c can be issued and revoked at any time, providing its issuer x fine-grained control of which other computations y can communicate with x.

- **Defensive message interpretation:** The evaluation semantics of a message m sent via CURL c by a computation x is x-specific and determined by x. The computation x assigns a binding environment B to a CURL c when x creates c. However, other computations y can send arbitrary data to x via c; as a consequence, x must (and is responsible for) protect itself against potentially malicious closures contained in the message m. By offering the minimal functional capability to the evaluation of m, x can minimize the impact of a potential attack.

2.3.3 COAST Computations

A COAST computation x is the execution of a closure c in the context of an execution site (E, B). An execution engine E may impose technical limits and execution semantics such as CPU cycles, memory available, storage or network bandwidth. A binding environment B defines the functional capability of c by providing the functions and global variables available to c. The closure c can augment its functional capability by receiving binding environments in messages received from other computations.

2.4 The Motile/Island implementation

An architectural style is a named collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system [45]. Architectural styles are by no means linked to implementation details, although architectural frameworks and programming languages may reify an architectural style. For example, the Map-Reduce [15] architectural style is implemented by Hadoop[1] but there are no additional constraints imposed by the inverse relationship.

Motile/Island is a reference implementation of the COAST architectural style. It provides an implementation for the COAST architectural elements.

Motile is a single-assignment dialect of Scheme deliberately created for defining COAST computations and the messages and CURLs they exchange [21, 22]. Motile is the language in which closures are written for computational exchange. All Motile data structures are designed persistent and immutable to avoid shared memory problems [37].

[1] http://hadoop.apache.org/

2.4.1 Islands

An **Island** is a single, homogeneous address-space occupied by one or more Motile **islets** [23]. An Island represents the concept of a COAST execution host.

Islands are self-certifying [27] entities, each of them is associated to a public key, all their CURLs issued are cryptographically signed, and all communications among them are encrypted. An Island is instantiated with one or more execution engines, binding environments, and trusted islets that bootstrap the COAST system.

2.4.2 Islets

In Motile/Island, **islets** are implemented as actors [5], that is, computational agents capable of receiving and sending messages asynchronously. In addition to transmitting and receiving messages, each actor is also capable of conducting private computations and spawning new islets. An actor is initialized with a specific binding environment, thus, with a well-defined functional capability.

2.4.3 Compilation and Serialization

Motile is both a programming language and a Scheme-to-Scheme compiler [22]. In particular, the compilation process translates from the Motile language to Racket[2], another Scheme-based programming language. After compilation, Motile structures are **serialized** and **transmitted** to other islets. On the other end, the closure are **deserialized** and then received by a given islet. It is worth noting that all free variables transmitted to an islet are rebound to the binding environment associated to the CURL used for the communication.

[2]http://racket-lang.org/

Chapter 3

Capability accounting

As previously explained, a **capability** is an unforgeable reference whose possession confers both authority and rights to a principal. **Capability accounting** is the practice of producing and maintaining a record or statement of capability events relating to a particular period or purpose, as well as the examination of those capability logs, that is, how the produced records can be queried, correlated, and analyzed in order to help design, develop, operate, and maintain secure decentralized systems, and provide the means, when possible, to assign blame upon malicious acts, or actions that inadvertently, break or jeopardize the integrity of a system.

3.1 Capability events

A **capability event** is a single occurrence of a process applied or related to a capability [38]. Examples of capability events are creation, exploitation, revocation, transfer, and rejection. Implementations of capability accounting must define which capability events are to be considered, which activities wlll take place, and in which form. Section 4.1 defines the capability events that were used during this study. Section 4.2 explains the included activities, and the tasks performed.

3.2 Activities

Capability accounting encompasses a series of activities that regulate and articulate the means to acquire and analyze capability events in compliance with technical, organizational, business-related, and legal protocols and regulations. Capability accounting relies on a framework that emphasizes systemic auditing, that is, the sanctioned inspection of capability logs of one or more parties, and auditing policy, the manner in which capability logs are examined, correlated, and reported, and defines the potentially required additional actions (e.g. revoking a capability, emitting a system alert, refusing service to a specific party).

The end-to-end framework used throughout this research project is described below. It models all the required activities that enable this implementation of capability accounting but it can be used as well as a reference for future implementations. Modeling all the tasks involved using an end-to-end framework helps understand their responsibilities, organization and dependencies as well as provides a common language for this study. Each activity usually relies on information produced by one or more of the previous activities, thus, the order of execution needs to be taken into consideration, although it is not mandatory to execute the activities immediately one after another. The activities included in the proposed model are: **capture**, **transfer**, **storage**, **query**, **preparation**, **evaluation**, **reporting**, and **action**.

3.2.1 Capture

Capturing involves all the considerations related to how, when, and where capability events are captured. It describes the technical, organizational and privacy-aware protocols and policies to capture information. For example, what information can be technically and legally captured? Which points in the software systems need to be instrumented or monitored to acquire the required information? Is the information available for acquisition all the time or only in certain periods? Is it feasible (i.e. possible without pushing the underlying system beyond acceptable execution parameters such as performance and latency) to obtain all the

capability events? Are there mechanisms that need to be put in place to guarantee capturing feasibility? Is there any transformation in the acquired events that needs to be applied?

3.2.2 Transfer

This activity describes the mechanisms used to ship out the acquired information. This is especially important when platforms have to be instrumented for capturing. Some of the questions that need to be addressed during this activity are: How is information routed to storage upon capturing? How is data serialized in order to be transferred? Is it necessary to wrap or represent the data in any specific manner? Which mechanisms are to be placed to enable the consumption of the transferred data?

3.2.3 Storage

The considerations during this activity are related to how capability events are stored for inspection. It includes the technical decisions to enable storing events as well as legal and privacy compliance, and the integration with the consecutive activities: transferring and querying. Some of the questions to address are: What database system(s) are to be used for storing capability events? Is it required to encrypt the stored information? In that case, what are the encryption parameters (e.g. algorithms, keys)? Are there pieces of data that cannot be stored for privacy or legal reasons, or that need to be stored separately? Are there physical or geographical constraints to store the information? Are there replication, performance or other non-functional requirements for storing capability events? What mechanisms are to be in place for retrieving the stored information?

3.2.4 Query

This activity defines where capability events are obtained or retrieved from, the manner in which the information is accessed, and the timing and frequency for these actions. Addition-

ally, the technical, legal and organizational parameters for obtaining the data as well as the use of database APIs are considerations relevant to this activity. For example, if capability events are stored in a database, this activity will be responsible for polling the data from it, and defining how often it will poll the database. Instead, if capability events are not stored (real-time analysis), this activity will include setting up communication mechanisms to subscribe for the events, or to enable other parties to send the captured events. Additional concerns that this activity must address are authentication and any other requirement to access the data, and data consolidation.

3.2.5 Data preparation

During this activity capability events are processed to be prepared for examination. Sub-activities involved during this task are:

- **Data selection:** Defining which capability events are to be included in the analysis, and which ones must be ignored. In addition, choosing what information from each capability event has to be considered.
- **Aggregation:** Characterizing how capability events must be combined or grouped to produce more complex structures.
- **Correlation:** Establishing relationships between events.

Data preparation may be coupled with the business logic behind a system evaluation. Let us say that performing a system action a (e.g. accessing a peer's service) produces a capability event e, as a side effect. Imagine it is necessary to verify that the action a happens no more than n times within t seconds. During the data preparation activity, the number of capability events e seen within t seconds will be grouped into a structure such as a list or a counter, that is, a structure more suitable for the information essentially required.

Therefore, this activity may also include algorithms, mechanisms and business logic to model events and prepare them for evaluation or visual representation.

3.2.6 Evaluation

The evaluation activity is likely to be the most representative in verification-based analysis. Tasks performed during evaluation aim to verify whether the capability events seen satisfy a given criteria. That is, this activity embodies the comparison between observed events, either in raw format or combined into more complex structures, and a comparable, expected entity. The evaluation encompasses the analysis and interpretation of incoming data to verify whether the system's execution meets any form of requirements desirable or mandatory (e.g. performance, latency, legal regulations, organizational business rules).

Designing the evaluation activity includes the definition of its algorithms and parameters. For example, imagine that capabilities in a system are to be transferred from peer A to peer D but with certain constraints. That is, it is acceptable that capabilities x,y and z are shipped from A to D, as long as they pass through B or C. The data preparation activity receives all capability events related to CURL transfers and correlates them forming lists of related events. The evaluation activity receives paths of exchange (i.e. lists of peers $P_1...P_n$ where a set of capabilities passed through) and verifies whether they satisfy the given criteria. In order to do so, it is required to define 1) the algorithm to verify whether a path (P_1, P_2, P_3) is correct (i.e. P_2 must be a valid intermediate peer), and 2) the parameters for performing such verification (i.e. define which are the valid peers: if $P_1 = A$ and $P_3 = D$ then $P_2 = B$ or $P_2 = C$).

3.2.7 Reporting

Reporting includes the generation of representations for the collected data and evaluation results. Some of the questions that must be addressed during this activity are: what information has to be reported? Which representation methods are to be used? Where must information be made available? Which mechanisms must be put in place so that the interested and authorized principals can get access to the reports? Are reports only temporary or do they need to be stored for future examination?

3.2.8 Action

During this activity, responses upon collected and analyzed information, are to take place. That is, responses depending on the collected data and the results of the performed evaluations. Those responses can take place within the software under monitoring, within the whole information system (i.e. hardware, software, process and people), at the intra- or inter-organizational levels or on an external systems. Furthermore, responses do not necessarily need to be automatic; a valid reaction could be a notification for software developers that a piece of code is malfunctioning and needs to be fixed or checked.

Reactions within the examined software refer to COAST-based actions that aim to adjust the parameters of execution of the monitored software, particularly, actions to prevent or mitigate harmful effects, or to assure that the system continues behaving as expected. Examples of possible actions at this level are the revocation of CURLs, the deployment of computations (agents) to correct a specific situation, and the replacement of computations to change the current behavior.

Responses related to the entire information system affect either the hardware, software, people or process supporting the information system as well as the monitoring system itself. For example, if an evaluation reports that the instrumented parts of the system under monitoring are impacting negatively the system performance, a viable reaction could be changing the instrumentation method (e.g. sampling events instead of capturing all of them) or disabling instrumentation in some of the points (potentially limiting some analyses, as discussed in Section 8.2). Additionally, the interaction between users and the software, and how data is manually or automatically prepared for input may need to be adjusted.

The activities listed in this section aim to provide a guideline for capability accounting implementation but are not shaped as a formal model. The intention is to create awareness of the activities that are very likely to take place, highlight dependencies between them, and raise many of the questions that analysts should address.

3.3 Exploratory and evaluation-based analyses

The proposed framework enables more than one venue for analysis. Depending on the nature of the process to model or the information to obtain, some framework activities are to be included or omitted, and each activity included will take place on a specific form. This research study focuses on evaluation-based analysis, that is, an expected behavior exists, and capability accounting is used to determine whether a system under inspection behaves in that manner. It is known in advance that the execution of the system should include specific actions and interactions, and that particular capability events should be generated as a side effect. Even if the capability events are not completely known, at least, some information about them can be anticipated. Therefore, they can be somehow compared with the observed events. Section 4 describes the implementation of evaluation-based capability accounting used throughout this work.

Another possible implementation of capability accounting is to enable exploratory analysis. The difference with the evaluation-based analysis is that, when exploring a system, there is no certainty about how the system should behave. That is, either there is no knowledge about the system or there are some indications that want to be confirmed by *seeing* the dynamics of the system: Who is communicating to whom?, with which frequency?, are there computations accumulating capabilities? are there capabilities being excessively revoked at any point?

Exploratory analysis can be used to observe, understand, and help create models than later on will be implemented using evaluation-based capability analysis [7].

An exploratory analysis based on capability accounting would make use of the *capture*, *transfer*, *storage*, *query*, and *data preparation* activities; because there is no comparison available, the *evaluation* would be omitted, and *reporting* would take place to present the findings. The *action* activity may trigger responses, although it is unlikely that they can be automated.

Chapter 4

Implementation of Capability accounting for evaluation

This chapter covers the full technological stack used to implement capability accounting for evaluation. It provides a comprehensive view of all the activities involved, the tools that were used, and the ones created. The entire implementation is aligned with the framework proposed in Chapter 3.

4.1 Capability events

Capability events are the principal unit of analysis of capability accounting. The capability events considered for this study are **creation**, **exploitation**, and **transfer**. Creation occurs when a COAST peer creates a CURL which allows other peers to send messages to it. Exploitation occurs when one peer sends a message to another peer via a CURL. Transfer occurs when a peer sends a CURL embedded in a message to another peer. In this implementation, capability events are represented as a map of key/values. Section 4.3.1 details which fields are to be considered for analysis.

4.2 Activities

The activities included for capability accounting and the tasks executed within their context depend on the kind of information to obtain and the analysis to perform. This implementation is evaluation-based, that is, the goal is to compare observed capability events with expected patterns modeled beforehand.

Capture is performed by an instrumented version of the Motile/Island platform. **Transfer** is carried out by a Racket STOMP [1] module added to the Motile/Island platform, the RabbitMQ[1] message broker, and a tool that connects RabbitMQ with MongoDB[2]. **Storage** is performed by the NoSQL MongoDB database. **Query**, **data preparation**, **evaluation**, and **reporting** are implemented by COMET. Next section explains how these tools were integrated and used.

4.3 Tools for capability accounting

Figure 4.1 illustrates all the tools that were used for implementing capability accounting, and how they interact with each other. Below, a description of all of those tools and their connections is provided.

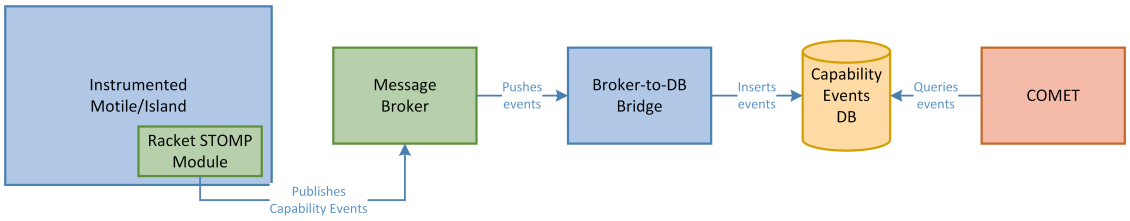

Figure 4.1: Overview of capability accounting for evaluation.

[1] http://www.rabbitmq.com/
[2] https://www.mongodb.org

4.3.1 Instrumented Motile/Island implementation

As mentioned in Section 2.4, Motile/Island is the reference implementation of the COAST architectural style. In order to capture the capability events required for this study, a few probes were added in specific pieces of code. Those *small* routines capture when 1) an Island is **started**, 2) a **new CURL** is created, 3) a message is **sent** via a CURL, 4) a message is **received** via a CURL, and 5) a CURL is **transferred** as part of the content of a message. After acquiring the events, those routines pass them to the *Racket STOMP module* to be shipped out.

Each captured capability event contains up to 7 fields:

- **source-island:** The *Island* who originated the capability event.
- **source-islet:** The *Islet* who originated the capability event.
- **time:** The *Island's time* when the capability event was produced.
- **curl-id:** The *ID* of the *CURL* associated to the event.
- **place:** Whether the communication was within the same Island (i.e. *intra*) or between different Islands (i.e. *inter*).
- **type:** The capability event type: *start-island*, *curl-new*, *curl-send*, *curl-receive* or *curl-transfer*.
- **mq-time:** The *time* when the event arrived at the *Broker-to-DB tool*.

Table 4.1 details the captured fields for each event type.

Depending on the type of analysis to perform, some fields will need to be used but others may need to be ignored. For example, in COAST, CURLs are unforgeable and tamper-proof, thus, if a computation x receives a message m from another computation y, x can trust the *curl-id* included in m because the CURL includes cryptographic information. However, the

Capability event	source-island	source-islet	time	curl-id	place	type	mq-time
island-start	X		X			X	X
curl-new	X	X	X	X	X	X	X
curl-send	X	X	X	X	X	X	X
curl-receive	X	X	X	X		X	X
curl-transfer	X	X	X	X	X	X	X

Table 4.1: Information available in capability events.

time field may need to be carefully interpreted because it cannot be guaranteed that the sender did not forge it.

The size of Motile/Island (COAST version *b20150315*), measured in SLOC, including tests and library bindings is 36457. The size of the instrumented version of Motile/Island is 37134 SLOC. Although no systematic tests were performed to compare performance in both implementations, the penalty in the instrumented version was not noticeable.

4.3.2 Racket STOMP module

The STOMP module was written in Racket 6.2 based on *racket-stomp 3.2*[3]. It implements STOMP 1.1 [1], and it is integrated in the instrumented Motile/Island platform. The Racket STOMP module is responsible for connecting to the RabbitMQ message broker and sending the captured events.

4.3.3 Broker-to-DB bridge

The Broker-to-DB bridge is a tool written in Java 7, based on Apache Camel[4], that subscribes to RabbitMQ to receive events from a queue using the AMQP protocol [5], and inserts them in a MongoDB collection.

[3]https://github.com/tonyg/racket-stomp
[4]http://camel.apache.org/
[5]http://www.amqp.org/

4.3.4 RabbitMQ

RabbitMQ is an enterprise reliable messaging system. It is used to route events from Motile/Island to MongoDB. Other consumers of capability events can connect to it to perform other analyses.

4.3.5 MongoDB

MongoDB is a NoSQL, high-performance database used to store capability events for future analysis.

4.3.6 COast Monitoring Event Tool (COMET)

COMET is a tool written in Java 7, responsible for retrieving capability events from MongoDB, and evaluating whether they match an execution model or not. It offers various techniques and operators to process capability events. Details about COMET and the features it offers are provided in Chapter 5.

Chapter 5

COast Monitoring Event Tool (COMET)

As part of assessing capability accounting, a tool called COast Monitoring Event Tool (COMET) was created to query, prepare, evaluate and report according to the capability events found in a database. COMET is an open-source application written 100% in Java that allows users to create assertions to verify the proper execution of a software system based on the observed capability events generated by the system under monitoring.

5.1 High-level architecture

The principal components in COMET are the Configuration Reader, Evaluation implementations, Evaluations Manager, and Query Handler, as illustrated in Figure 5.1.

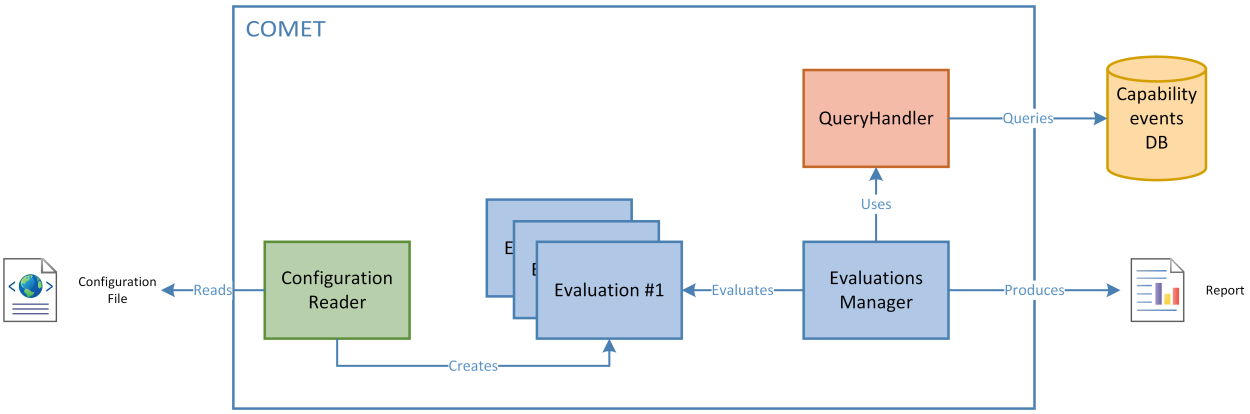

Figure 5.1: COMET High-Level Architecture.

5.1.1 Configuration Reader

This component is responsible for reading and parsing an XML configuration file that contains parameters for execution as well as the definition of all of the evaluations that are to be performed.

5.1.2 Evaluation Implementations

An evaluation represents how data has to be prepared for analysis and the logic to verify whether the observed events correspond to the expected events. Implemented evaluations are listed in Section 5.2.2.

5.1.3 Evaluations Manager

The main responsibilities of this component are the orchestration of the evaluations, that is, preparing the evaluations to be launched and controlling their execution as well as showing the results to the user.

5.1.4 Query Handler

A Query Handler is a database-dependent component that enables the retrieval of capability events from a database.

5.2 Configuration

COMET provides, out-of-the-box, a self-documented XML file that can be used as a template for custom configuration files.

The configuration file includes two main sections: global properties and assertions. The former defines all the properties that are needed for the entire analysis such as database credentials, and global parameters for all the evaluations. The latter defines all the verifications

that will be performed and their parameters of execution.

5.2.1 Global properties

Property	Definition
mongo-host	Host where the MongoDB instance that contains the capability events is running.
mongo-port	Port where the MongoDB instance that contains the capability events is running.
mongo-db	Database where capability events are stored.
mongo-collection	Collection where capability events are stored.
last-component	Because the capability events logged in the database can span multiple executions of the system under examination, this property defines the starting point for analysis. It basically defines, as a starting point, the time when a specific component was launched.
correlation-field	For assertions where the order of capability events is considered, this field is used to determine whether an event appeared before another.

Table 5.1: Global properties in COMET configuration.

For example, if *last-component* is *trader* and *correlation-field* is *mq-time* then the starting point of the analysis will be the last time when the *trader* was launched, and all capability events whose *mq-time* is greater than that, will be included.

5.2.2 Evaluations

An evaluation verifies whether the observed entities satisfy a given assertion. There are five possible results for this process:

- *PASS:* The observed entities satisfy the requirements posed by the assertion.

- *FAIL:* The observed entities do not satisfy the requirements posed by the assertion.
- *WARNING:* The observed entities either do not satisfy the requirements but that does not impose significant problems, or they do satisfy the requirements but with minor (often unimportant) differences.
- *ERROR:* The evaluation could not be performed because an error (e.g. an exception) occurred during the process.
- *UNKNOWN:* Either the evaluation could not be started or it was not able to arrive at a result.

COMET implements different types of evaluations to correlate and analyze capability events: **sequence**, **volume**, **match**, **when**, **exists**, **not**, **unordered**, and **or**. Most of them can be combined to provide more flexibility.

Sequence evaluation

Sequence evaluates whether capability events with certain properties were or were not seen, and the order in which they should or should not have appeared.

Each sequence is defined as a series of ordered capability events. For each capability event, it is possible to define the values of the fields that are to be present. Additionally, COMET allows the use of wildcards to connect different events. The following is an example of a sequence assertion:

```
<sequence description="Island A ships a new inter CURL.">
    <event description="Island A creates a new inter CURL.">
        <source-island>A</source-island>
        <type>curl-new</type>
        <place>inter</place>
        <curl-id>$capture:curl-A</curl-id>
    </event>
    <event description="A transfers the new inter CURL.">
        <source-island>A</source-island>
        <type>curl-transfer</type>
        <curl-id>$read:curl-A</curl-id>
```

```
    </event>
</sequence>
```

The previous sequence assertion is compounded by two event patterns that had to be seen in the given order within the analysis period to satisfy the assertion.

The first event defines a capability event that represents the creation (*type: curl-new*) of a CURL for *inter-island* communication in Island *A*. The *$capture:curl-A* value specifies that the value present in *curl-id* has to be temporarily stored under the *curl-A* key.

The second event defines a capability event representing the transfer of the CURL created in the previous event. It specifies that Island *A* is shipping out (*type: curl-transfer*) a CURL whose *curl-id* is the one stored under the *curl-A* key.

Volume evaluation

Volume evaluates whether a capability event matching certain properties is seen between n and m times within a specific time period. The following is an example of a volume assertion:

```
<volume description="Island A uses the service represented by CURL abc between 5 and 10
 ↪  times per second" minrange="5" maxrange="10" timerange="1" unit="seconds">
    <event description="Island A uses service referenced by abc">
        <source-island>A</source-island>
        <type>curl-send</type>
        <place>inter</place>
        <curl-id>abc</curl-id>
    </event>
</volume>
```

In the previous example, the given event pattern represents that a computation running in Island *A* sends (*type:curl-send*) an inter-island (*place:inter*) message using a CURL whose ID is *abc*. Capability events matching this pattern must appear between **5** and **10** times within **1 second**.

Match evaluation

The *Match* evaluation verifies whether a previously seen event satisfies certain criteria. In order to identify events, the attribute *capture* must be declared in an *event* evaluation, and used as *captureKey* in the *Match* evaluation. The following is an example of this type of evaluation:

```
<event description="A computation in Island A created a CURL." capture="new-curl">
    <source-island>A</source-island>
    <type>curl-new</type>
</event>

<match description="CURL is xyz" captureKey="new-curl">
    <event>
        <curl-id>xyz</curl-id>
    </event>
</match>
```

In this example, the *event* evaluation passes when a computation in the Island A creates a new CURL (*type:curl-new*). That event is temporarily stored under the *new-curl* key (*capture="new-curl"*). When *Match* is evaluated, it looks up the event under the *new-curl* key and compares it with the fields declared under *<event>*, in this case, *curl-id=xyz*.

When evaluation

The *When* evaluation is compounded of a condition and a predicate. It verifies that every time the condition (a list of evaluations $[e_0...e_m]$) passes, the predicate (a subsequent list of evaluations $[e_n...e_z]$) must pass as well. The number of evaluations in the condition is determined by the *conditions* attribute.

```
<when description="Island A only exploits CURL xyz" conditions="1">
    <event description="Island A sends a message via c" capture="send-event">
        <source-island>A</source-island>
        <type>curl-send</type>
    </event>

    <match description="CURL is xyz" captureKey="send-event">
        <event>
            <curl-id>xyz</curl-id>
```

```
        </event>
    </match>
</when>
```

Exists evaluation

The *Exists* evaluation is similar to the *When* evaluation with the difference that *Exists* passes if the pattern is matched at least once, instead of every time.

```
<exists description="Island A only exploits CURL xyz" mandatory="1">
    <event description="Island A sends a message via c" capture="send-event">
        <source-island>A</source-island>
        <type>curl-send</type>
    </event>

    <match description="CURL is xyz" captureKey="send-event">
        <event>
            <curl-id>xyz</curl-id>
        </event>
    </match>
</when>
```

The example is the same as in the *When* evaluation. However, in this evaluation, it passes if and only if the condition and the predicate pass at least once. The size of the list of conditions is determined by the *mandatory* attribute. The rationale for this example is that Island A must send a message via CURL xyz at least once.

Not evaluation

The *Not* evaluation changes the result from an inner evaluation. The *Not* evaluation must have only one nested evaluation. Table 5.2 shows the modifications applied by *Not* to the result of the inner evaluation.

Inner result	***Not* result**
PASS	FAIL
FAIL	PASS
WARNING	PASS
ERROR	ERROR
UNKNOWN	UNKNOWN

Table 5.2: How the *Not* evaluation transforms the result of its nested evaluation.

Unordered evaluation

The *Unordered* evaluation verifies that all of its inner evaluations pass, regardless of the order.

```
<unordered description="Island A and B send messages.">
    <event description="Island A sends a message">
        <source-island>A</source-island>
        <type>curl-send</type>
    </event>
    <event description="Island B sends a message">
        <source-island>B</source-island>
        <type>curl-send</type>
    </event>
</unordered>
```

The *Unordered* evaluation passes if and only if the two events were seen regardless of which one appeared first.

Or evaluation

The *Or* evaluation verifies that, at least, one of the inner evaluations pass.

```
<or description="Island A or B send messages.">
    <event description="Island A sends a message">
        <source-island>A</source-island>
        <type>curl-send</type>
    </event>
    <event description="Island B sends a message">
        <source-island>B</source-island>
```

```
8           <type>curl-send</type>
9       </event>
10  </or>
```

The *Or* evaluation passes if Island *A* or Island *B* sends a message.

5.2.3 Output

Final outcome displays the result of the entire execution and breaks down all the evaluations showing the individual results and info associated to them. As evaluations can be nested, the format of the output is displayed as a tree:

```
[FINAL RESULT] <Overall description>. Type: Unordered Message: <Extra information>
    [RESULT 1] <Description of evaluation 1> Type: <Evaluation type> Message: <Extra
    information>
        [RESULT 1.1] <Description of evaluation 1.1> Type: <Evaluation type> Message:
        <Extra information>
        [RESULT 1.2] <Description of evaluation 1.2> Type: <Evaluation type> Message:
        <Extra information>
    [RESULT 2] <Description of evaluation 2> Type: <Evaluation type> Message: <Extra
    information>
        [RESULT 2.1] <Description of evaluation 2.1> Type: <Evaluation type> Message:
        <Extra information>
    [RESULT 3] <Description of evaluation 3> Type: >Evaluation type> Message: <Extra
    information>
```

The **message** field is primarily used to display contextual information when an evaluation fails. For example, if an event is not found, the following message will be displayed:

```
[FAILED] <Description of evaluation>. Type: Event Message: Event [type:curl-send (EQ),
source-island:A (EQ), curl-id:xyz (EQ), mq-time:1439083845520 (GE)] could not be found.
```

The format to describe an event is a set of fields defined as $key : value(operator)$. The operator can be equal (EQ), greater (GT), greater or equal (GE), lower (LT), lower or equal (LE), or not equal (NE).

The default type of the root evaluation is *Unordered*.

Chapter 6

Electronic trading and software systems

Decentralized systems can be found in several domains such as healthcare, e-government, emergency-response, army, and electronic trading [23, 21]. The basic nature of such systems is that they are distributed systems in which components evolve with little or no coordination, and they are designed, developed, operated, and maintained by different organizations.

Electronic trading has several parties involved, including individual traders, trading firms, brokerage firms, exchanges, gateways, dark pools, Alternative Trading Systems (ATS), infrastructure firms and a myriad of systems to support traditional trading and High-Frequency Trading (HFT). Communications among systems is dictated by open [14, 34], and by proprietary protocols [17] that must be integrated and adapted by most of the systems involved. Illustratively, MagniFIX[1]©, a monitoring suite developed by CameronTec Group[2], one of the lead companies developing infrastructure for the Capital Markets industry, currently provides support for more than 60 different protocols, and informs upcoming support for 37 more [24]. Some of those protocols have been standardized but most of them are proprietary.

In addition to the number of different venues that can be used to trade (e.g. exchanges, dark pools, ATS), systems can interact in very diverse ways, causing the impact of a single change to spread across several other systems. For example, a Trading Firm can connect

[1]http://www.greenlinetech.com/products/magnifix.php
[2]http://www.camerontecgroup.com/

directly to an exchange via a DMA[3] connection [2] or through a broker. The broker, likewise, provides financial services to their clients but also invests their own assets in potentially multiple exchanges. Consequently, exchanges receive connections from traders and brokers, but also from HFT[4] firms. Market data generated by an exchange is consumed by all the previously mentioned parties but also by monitoring and infrastructure systems, and from systems in other trading venues who use their information as a reference. These examples are intended to provide a notion of the complexity and diversity of electronic trading interactions.

6.1 Domain relevance

The introduction of this chapter explains why systems in the financial domain are considered decentralized systems, and illustrates some of the platforms, systems and components involved in electronic trading. This section highlights the importance of the domain in the global economy, and depicts why it is especially important that software systems are secure.

6.1.1 Global trading volume

The adoption of electronic platforms for trading has been increasing and expanding worldwide, providing more efficiency and transparency to financial markets, and dramatically raising trading volume and liquidity.

There are endless financial instruments that can be traded ranging from stocks, bonds, and corporate debt, to futures, contracts, and options, just to name a few.

The World Federation of Exchanges[5] groups the major exchanges around the world, and produces statistics that are available to the general public about their members. According to their information, the value of shares traded by their members globally between January and June of 2015 accumulated almost 58 trillion dollars [36], that is more than three times the

[3]Direct-Market Access: a point-to-point connection to an exchange.
[4]High-Frequency Trading
[5]http://www.world-exchanges.org/

GDP of United States, the world's largest national economy, in 2014 [19]. The same source reported that the trade of bonds accumulated 8 trillion dollars within the same period. Additionally, the Aite Group[6], a very well-known consulting firm, reported that the average daily volume of currencies traded was $5.5 trillion in 2014 [18]. It is worth noting that the reported information only includes equities, bonds and currencies but does not include other securities such as derivatives.

It is easy to see how impactful software systems are in such large economy. Obviously, electronic trading is mostly controlled by software, and its lack of security or integrity can be disastrous for a company or even for the entire industry.

6.1.2 Major incidents in financial trading

Trading systems are not free of software issues, and it is not surprising after reading the enormous sum of money involved in trading, that those issues may have a tremendous impact on the affected companies and markets.

On July 8, 2015, the New York Stock Exchange (NYSE) suffered an "internal technical issue" [35] that forced them to stop trading activities for almost four hours. Although losses were not published, it prevented millions of dollars from being traded within that period. Additionally, the FBI and the Department of Homeland Security were involved to determine if it was a cyber-attack. After investigation, the reported root cause was a configuration issue [20].

On August 20, 2013, a software bug in Goldman Sachs' systems that set wrong price limits made the company flood American exchanges with stock-option orders. Although exchanges reviewed and rolled back many of those orders, Goldman had to face losses for more than 100 million dollars [26]. The company did not provide details about the problem but admitted that it was a programmatic error.

On August 1, 2012, Knight Capital deployed untested software that executed a number

[6] http://www.aitegroup.com/

of orders that were supposed to be done over a period of days, in less than an hour. The company lost $400 million and had to be acquired by investors who paid almost half a billion dollar for it. In this case, a programmatic error almost caused a company to shut down.

On May 6, 2010, a "flash crash" caused the Dow Jones Index (DJI) the biggest intraday point decline in its entire history (about 9%) [28], but it recovered almost to normal in around half an hour, as seen in Figure 6.1.

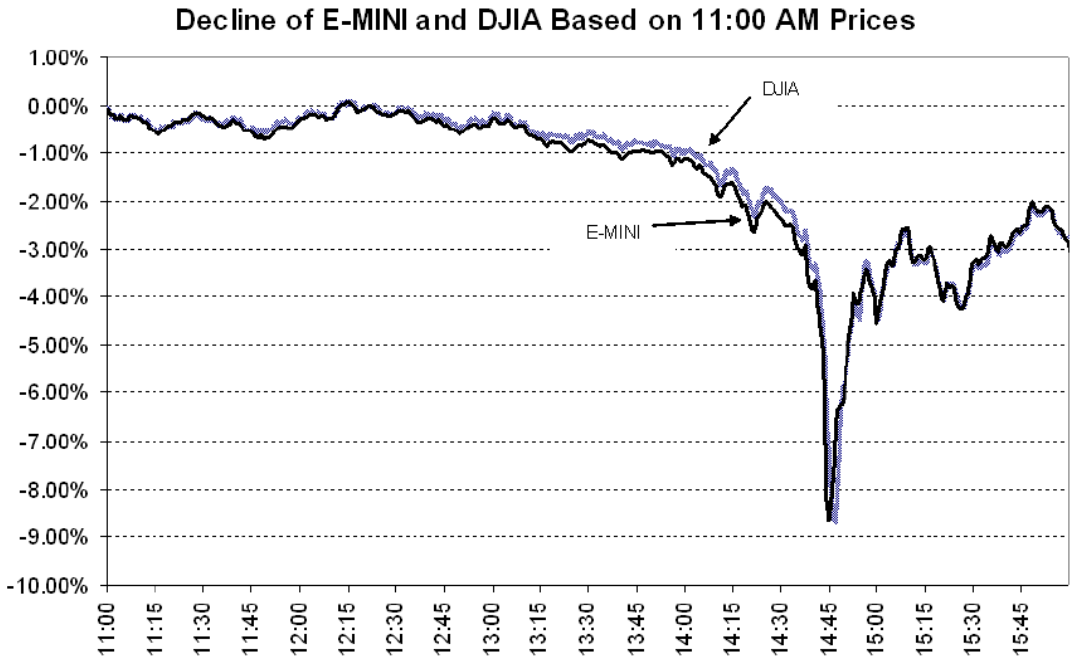

Figure 6.1: Impact on prices of E-Mini S&P and Dow Jones Industrial indexes caused by the Flash Crash in 2010. [42]

The main difference between the "flash crash" case and the previous ones is that this one was not caused by a software bug. After five years of investigation, the Commodity Futures Trading Commission (CFTC) concluded that Navinder Singh Sarao, a British high-frequency futures trader, "was at least significantly responsible for the order imbalances" in the derivatives market which affected stock markets and exacerbated the flash crash [12]. The article's authors highlighted an essential reflection about how electronic trading is evolving that reinforces the motivation of this work:

"Sarao didn't cause the flash crash single-handedly, authorities say. [...] Regulators initially concluded that a mutual fund company – said to be Waddell & Reed Financial Inc. of Overland Park, Kansas – played a leading role. Many in the industry countered that a confluence of several forces, including high-frequency trading, was probably behind the crash. By all accounts, the flash crash was more than a mere technical glitch. It raised fundamental questions about how vulnerable today's complex financial markets are to the high-speed, computer-driven trading that has come to dominate the marketplace."

6.2 Challenges on electronic trading software systems

The last sections show that software systems in the financial industry are decentralized systems, highlight the relevance of the domain in the global economy, and list the most important incidents and their effects in the last five years. It is clear at this point the impossibility of guaranteeing security and integrity in these systems, and that a programmatic or configuration error can have disastrous consequences.

My former job as a professional developer and Tier III support (in a three-tiered technical support model) exposed me to a variety of challenges present in software in the financial industry. Particularly, **inspection, hot updating, debugging**, and **forensic analysis** are activities that must take place but are exceptionally problematic because of the characteristics of the environments in which the systems operate.

Security and privacy concerns prevent the remote operation of systems, even by the vendors who developed and helped deploy them. As an ironically comic example, the way to inspect a running software in one of the top-three world's largest banks, offering financial services in New York was the following (see Figure 6.2): an in-site vendor employee shared the screen of a terminal to another vendor employee working in a remote office, using the credentials of a bank employee who visually monitored the whole process. The terminal was connected to one of the servers running the software under inspection. The remote operator

had to ask the in-site operator to run every command through SSH, and saw the response via the shared screen on the terminal. In some occasions, another hop had to be added if the server to debug was in a region different from New York. In summary, inspecting the execution of a system could take three people and three levels of indirection.

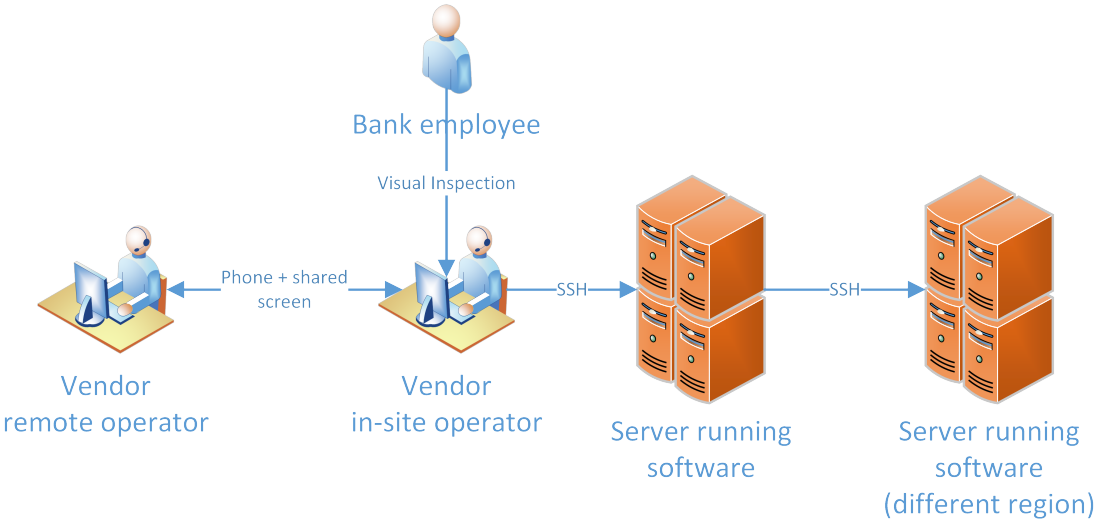

Figure 6.2: Challenges troubleshooting issues in the environment of an electronic trading system.

Although execution environments host server-class computers, the hardware and software platforms differ considerably among companies. As a result, infrastructure and application systems must be adaptable to operating systems such as Linux, Windows, FreeBSD, Solaris x86, and Solaris SPARC. Switches, routers and Network Interface Cards (NIC) cause different latencies and throughput that systems must account for. Available RAM memory and storage vary, making it difficult, and sometimes impossible, to calculate in advance how much information a buffer can hold; worse still, the nature of that information changes, that is, size and representation of messages depend on each particular market. Consequently, deployed systems must be completely parameterizable to adjust to custom environments. However, in addition to the security and privacy situations aforementioned, financial systems must be zero-downtime, meaning that they must be prepared to gracefully handle every single change that may potentially need to be applied during a trading session. Continuous monitoring

is, again, essential for this purpose. Software has to be monitored before, during, and after changes are applied to guarantee its proper execution.

Another challenge of electronic trading systems is diagnosing when they do not behave as expected. Systems cannot be directly accessed or restarted, as previously explained. Many times, because of privacy resons, vendors are unable to test their software with real data, even if both companies signed a non-disclosure agreement (NDA). The only alternative that developers have to diagnose a problem is analyzing the log files produced by their applications. However, two salient issues arise: First, developers must be extremely cautious on how they instruct systems to log information; too much logging penalizes the system performance and may eat up the available storage, but too little logging may jeopardize finding the solution to a problem. To complicate the situation even more, some companies do not allow log files to be shipped out of the running environment because they might contain sensitive information, therefore, vendors must analyze the log files produced by their applications remotely connected (as shown in Figure 6.2) to the execution host.

Capability accounting allows monitoring and verification to ensure the proper execution of software systems, and can help software developers to localize faults without accessing private information, as shown in the evaluation of this work. As electronic trading systems become more autonomous overtime, the chances for a human operator to stop them when something does not behave as expected are much lower but consequences are higher, as seen in the latest major incidents. This fact motivates the research of practices that reduce the risk of designing, developing, and deploying software vulnerable to attacks and human mistakes. Even assuming that mitigating those risks will ever be possible, state-of-the-art research and technology is far from solving present challenges: security, integrity, adaptability, differentiated service to clients, and rapid implementation. Nevertheless, it is worth investigating techniques that minimize risks, and enable and encourage continuous monitoring.

The present work explores the use of capability accounting with the COAST architectural style in the electronic trading domain. COAST and capability accounting complement

one another by providing the principles to build secure and adaptable systems from the ground up, while enabling continuous monitoring through architecture to verify their proper execution.

Chapter 7

Evaluation

Chapter 3 presents the concept of capability accounting and a framework that includes the fundamental capability elements and a series of activities to guide how those elements are to be handled. Chapter 6 highlights the importance of the financial trading domain in the global economy, and why systems in electronic trading qualify as decentralized systems. Additionally, it presents a list of software-related incidents that caused millions of dollars in losses.

This chapter presents an assessment of capability accounting as technique, that is, it addresses the question *is capability accounting a well-suited form of architectural accountability?*, and verifies that the framework and techniques proposed can help design, develop, operate, and maintain secure decentralized systems. With these goals in mind, I created a COAST-based prototype of a financial trading system that reflects the interactions of systems from various organizations participating in electronic trading operations. Even though real-world systems face challenges that are not present in the created prototype such as performance, low-latency, and data volume; the prototype illustrates how capability accounting can be used to overcome major challenges during development, deployment, operation, and maintenance of decentralized systems within the financial trading domain.

The prototype allows traders to deploy trading algorithms (i.e. trading computations) in

a host that executes them. As part of its operation, the trading computation receives relevant market information, and places buy and sell orders. I created various implementations of trading computations behaving in slightly different ways. The *base* computation reflects a normal execution, that is, the computation behaves as a trader would expect. All of the other computations were modified to introduce faults[1]. The intention was to produce computations that expose different failures. Because different faults can produce the same failure, some of the problems exposed in this section have multiple faulty computations associated; capability accounting is used to help localize these faults.

Below, a description of the prototype, the base computation and its trading strategy, and five problems with their faulty computations are presented.

7.1 An electronic trading prototype

The created prototype illustrates the actions and interactions between a Trader, a Broker, and an Exchange. The collaboration is carried out by systems presumably developed separately by the three participants.

The Trader develops a trading algorithm in-house but uses a Broker's infrastructure and connection to trade financial instruments (such as stocks) on an Exchange. The sequence of execution is as follows: the trading firm deploys its algorithm (i.e. a computation) to the Broker system, where it is executed, along with many others; the computation subscribes to the Risk Management and Market Data servers to receive risk and market updates, and places orders, based on execution parameters and custom business logic (e.g. prices, trading volume, risk changes) [28] with the Order Router (Figure 7.1), and receive execution reports.

[1]Here, *fault*, *failure* and *problem* follow the IEEE standard [9].

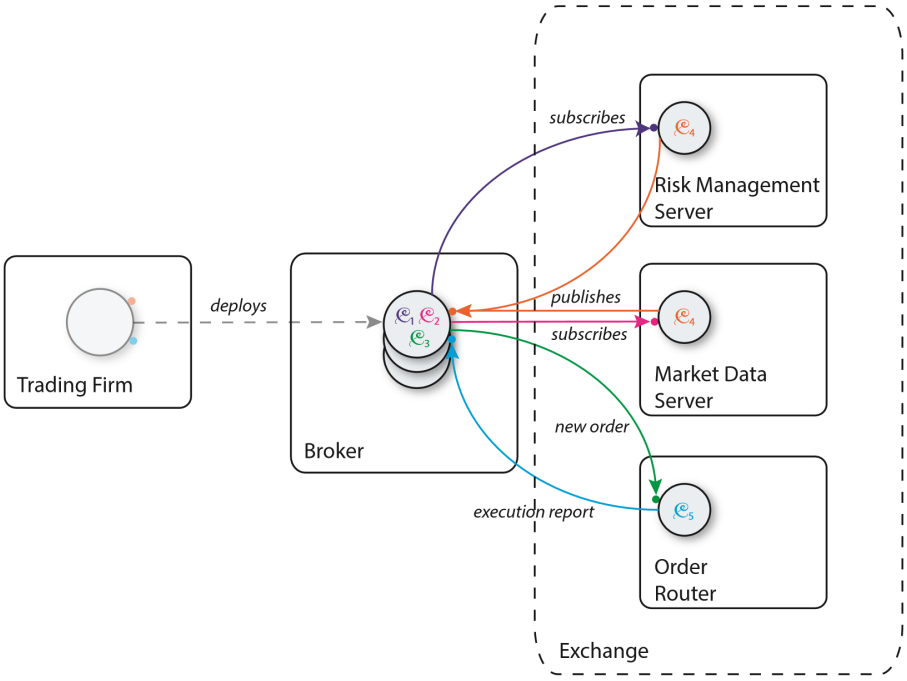

Figure 7.1: Prototype of a trading system including a Trader, a Broker and an Exchange.

7.1.1 Participants

- **Trader:** An individual who engages in the transfer of financial assets in any financial market, either for him- or herself, or on behalf of someone else [4]. It provides the *Trading Computation*.

- **Broker:** The firm providing financial services, computing infrastructure, and connections to exchanges for traders [43]. Any party with a direct market-access connection (DMA) [2] to an Exchange is able to trade on it; however, those DMAs are very expensive, thus, most traders use broker services to interact with Exchanges. It provides the *Trading Computations Host*.

- **Exchange:** The highly organized market where traders and brokers buy and sell financial instruments such as stocks, bonds, futures, and currencies [3]. It provides the *Risk Management Server*, *Market Data Server*, and *Order Router*.

7.1.2 Subsystems and components

- **Trading computation:** A piece of software representing an algorithm capable of performing trading actions such as placing buy and sell orders. It is deployed by the *Trader* in the *Broker*, and interacts with the *Risk Management Server*, *Market Data Server*, and *Order Router*.

- **Trading computations host (Broker):** Infrastructure provided by a *Broker* to support the execution of *Trading Computations*. It receives a *Trading Computation*, and provides computational resources for its execution. The terms *Trading Computations Host* and *Broker* are used indistinctly in this chapter.

- **Risk Management Server:** A component responsible for providing risk-related information such as the exposure of a financial instrument to volatility. *Trading Computations* receive risk-related data on a subscription basis.

- **Market Data Server:** It provides real-time market updates based on client subscriptions. Market data includes latest prices and trade-related information. *Trading Computations* subscribe to receive specific market updates.

- **Order Router:** It is the access point to place orders and receive execution reports. *Trading Computations* place orders on a given CURL, and must provide a reply-to CURL to receive execution reports.

The Market Data and Risk Management servers publish predefined updates. They have input files containing records and the delay that must be applied between each update. Those input files were intentionally created to give support to the implemented trading strategy, and more important, they allowed us to repeat the experiment in the exact same conditions as many times as needed. Admittedly, in real-world, trading computations do not have such knowledge in advance; nevertheless, I strongly believe it does not have an impact on the

result of this evaluation because the verification of execution does not take financial data into consideration.

In order to create the input files for the Market Data and Risk Management servers, we created tools that receive parameters as an input, and produce the described log files. As a result, it was possible to automatically create log files with arbitrary size, and reuse them in all the experiments.

7.1.3 Prototype's technical information

The prototype was developed with COAST *build 20150315*, which uses Racket as the Execution Engine, running on Ubuntu 15.04 64-bit. The Racket version used was 6.2. COAST dependencies are packaged in the COAST distribution, thus, it is not necessary to download additional libraries.

The size of all the components included in the prototype are measured in SLOC, and shown in table 7.1. The number associated to a component includes the SLOC of all the source files related to it. Common code such as utilities used in multiple components is counted separately.

Component	**SLOC**
Trading Computations Host	239
Risk Management Server Host	177
Market Data Server	198
Order Router	204
Base Trading Computation	243
11 Faulty Trading Computations	2449
Common code	191
Total:	**3701**

Table 7.1: Size of the different prototype's components.

7.2 *Base* trading computation

The *base* trading computation works in the context of the created prototype; once deployed in the Broker, it interacts with the Risk Management and Market Data servers, and the Order

Router. It implements an arbitrary trading strategy and follows the contracts (i.e. how interactions should take place) defined by the Exchange's components. This computation works as expected, that is, it has been tested and did not show signs of faults.

This computation is the reference implementation that *works well*, and from which all the faulty computations are derived.

7.2.1 Trading strategy

An arbitrary trading strategy was implemented as part of the base computation to interact with the prototype's other components. It does not follow precise *standard* strategies [43, 6] because that is not relevant for the scope of this study. However, it does implement a basic scenario of *stop loss* [44], where a trader decides to sell its partial or total possession of a stock when it reaches certain minimum price.

The Trading Computation represents an investment in Google[2], Facebook[3] and Yahoo[4] stocks. The prices of one *GOOG* share is $642, *FB* is $95, and *YHOO* is $36, at the the beginning of the scenario.

The Trading Computation behavior depends on the market data and risk events it receives. As mentioned above, those events are provided by the Market Data Server and the Risk Management Server, respectively. The actual updates they send depend on their input files which were, for this study, intentionally modeled to support this scenario.

The Market Data Server was parametrized to send updates of *GOOG* (Google), *FB* (Facebook), YHOO (Yahoo), *AAPL* (Apple), *AMZN* (Amazon) and *MSFT* (Microsoft), although the last three companies were not part of the computation's subscription, thus, those events were ignored. Throughout the execution of this scenario, *GOOG* transactions increase or decrease its price by a random number between $0 and $2, that is, each transaction can only increase or decrease the *current* price by a number within that range. *FB*

[2]http://money.cnn.com/quote/quote.html?symb=GOOG
[3]http://money.cnn.com/quote/quote.html?symb=FB
[4]http://money.cnn.com/quote/quote.html?symb=YHOO

transactions can move the price between \$0 and \$2 as well. *YHOO* transactions can only decrease the price, and it is done in variations between \$0.4 and \$0.2, therefore, if the *current* price of a *YHOO* stock is p, the next price will fall between $p - 0.4$ and $p - 0.2$. It can be seen that, in the long term, *GOOG* and *FB* keep the same prices or might experience slight variations, whereas *YHOO* suffers from a steady price reduction. As a consequence, in this computation, *YHOO* is the stock that triggers the *stop loss* scenario.

The Risk Management Server was parametrized to send updates of *GOOG*, *FB* and *YHOO*, since those are the stocks in which the trader is interested in. Risk events are used by the Trading Computation to make decisions when there are multiple alternatives.

The trading scenario has four phases described as follows:

- **Accumulation:** The Trading Computation buys stocks of *GOOG* and *FB* at current price. *YHOO* is not traded but its price is falling.
- **Partial stop loss:** *YHOO* price has reached \$27, and the Trading Computation reacts by selling 50% of its possession.
- **Total stop loss:** *YHOO* price kept falling and reached \$23. The Trading Computation sells all its remaining possession of *YHOO*.
- **Redistribution:** The Trading Computation buys stocks of *GOOG* and *FB* using the money it obtained from the previous sells. The distribution between *GOOG* and *FB* is determined by the latest risk information about those stocks.

7.2.2 Interaction with the prototype

Trading Computations are the unit of exchange between the Trader and the Broker, are the subject of execution within the Broker, and during their execution, they interact with the Market Data and Risk Management servers, and the Order Router. Considering these actions and interactions is essential to understanding how the verification model works. Below,

a description of the interactions between components is included. A few considerations must be taken in order to interpret the presented sequence diagrams. Specifically, when an action is denoted between two components, it does not necessarily imply function invocation. For example, $send(c)$ must be interpreted as one islet making the action of *sending* the computation c to another islet using an available underlying mechanism for that, it is not that the sender is invoking the $send()$ function on the receiver. The following notations are used in the diagrams:

- $send(c)$: An islet x sends a computation c to another islet y via some CURL.
- $call(f, [args])$: An islet x invokes the function f present in its binding environment. If arguments $args$ are present, they are passed to the function f.
- $spawn(c)$: An islet x executes the computation c in the Island where it runs.

Trading Computation deployment

The deployment of a Trading Computation takes place in the Trading Computation Host offered by the Broker. The host offers a service ($server.registration@broker$) that receives computations to be deployed and executed. The Trader possesses the service's CURL that enables it to send its Trading Computation. The UML sequence diagram in Figure 7.2 illustrates how a computation c is sent from the Trader ($dispatcher@trader$) and executed at the Broker side. The computation c must be a Motile thunk, that is, a Motile function with no parameters. After receiving c, the Broker ($server.registration@Broker$) executes c on its own Island.

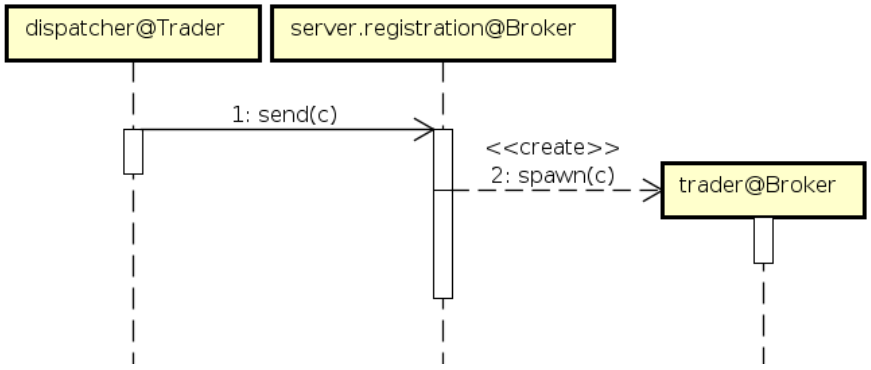

Figure 7.2: Deployment of a Trader's trading computation in the Broker's infrastructure.

Market Data and Risk registrations

Once the Trading Computation is deployed, it has to subscribe in the Market Data and Risk Management servers to receive market and risk updates required for its execution. Figure 7.3 details the interaction between the Trading Computation and one of the servers[5]. The computation executing on the Broker (*trader@broker*) ships another computation r to each of the servers. The computation r is executed at the server side, and registers

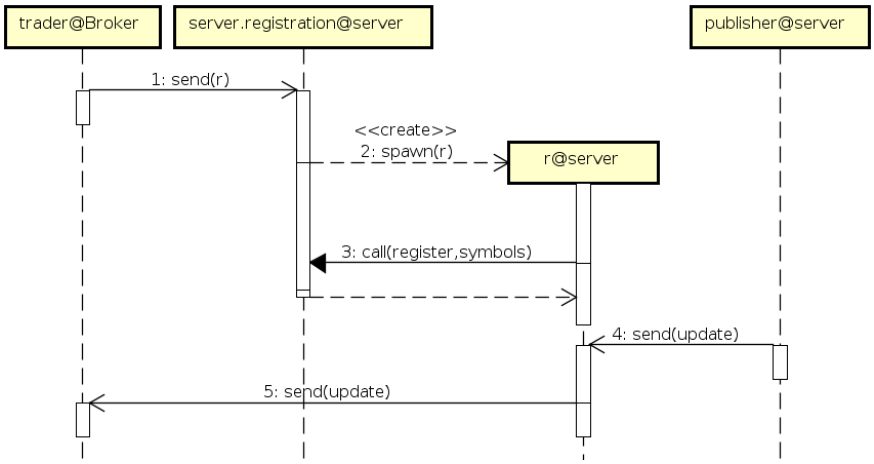

Figure 7.3: A trading computation running in the Broker subscribes to receive market and risk updates.

itself ($call(register, symbols)$) to receive the updates the (parent) Trading Computation is interested in. Each update is sent locally by *publisher@server* and received by r at the

[5]The interaction is identical for both servers

server side and forwarded to the Trading Computation at the Broker.

Placing orders

The Trading Computation running at the Broker (*trader@broker*) holds a CURL to place orders in the Order Router. Every time a new order x is to be put, the Trading Computation must 1) create a sub-computation (*order.x@broker*) to account for the order x during its lifespan, 2) receive a CURL (*curl.x*) from the sub-computation, and 3) send the order to the Order Router, along with the CURL received in the second step ($send(x, curl.x)$), that the Order Router will use to send execution reports back ($send(execreport.x)$). Figure 7.4 illustrates this sequence of actions and interactions.

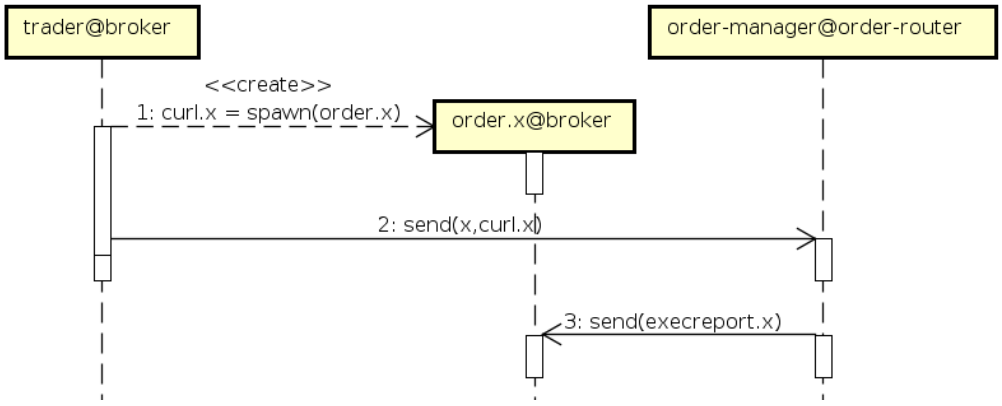

Figure 7.4: A trading computation running in the Broker places an order on the Order Router. As a result, it receives an order's execution report.

7.3 Verifying software execution

I created a model based on capability accounting in COMET to verify the proper execution of the base trading computation. In addition, a series of faulty computations were executed to assess that 1) COMET was able to detect when an execution was not flawless, and 2) COMET helped localize the capability event associated to the introduced fault.

As Figure 7.1 shows, the following actions and interactions are expected to take place in the prototype:

1. The trader ships and deploys a customized trading computation to the Trading Computations Host, running in the Broker.

2. Trading Computation subscribes to receive risk and market updates from the Risk Management and Market Data servers.

3. Trading Computation starts receiving updates.

4. Trading Computation places orders through the Order Router.

5. Order Router sends execution reports back to the Trading Computation.

In the context of this study, we assume that the execution of a trading computation is successful if none of the problems listed in Table 7.2 arises. Those problems fall within application, business and security categories.

1	Trading Computation does not place orders.	Application
2	Trading Computation does not notify the trader.	Application
3	Trading Computation opens a backdoor.	Security
4	Trader uses a different service level.	Business
5	Trading Computation places too many or too few orders.	Application

Table 7.2: Potential problems verified by COMET.

7.3.1 Verification model

COMET analyzes the collected capability events to evaluate whether specific events occurred or not, the order in which they appeared, and the frequency. The evaluations described in Chapter 5 were combined to create the verification model showed in figures 7.5, 7.6 and 7.7. The verification model encompasses an assertion of the proper deployment and initialization of the trading computation, the number of orders placed, and communications from a security perspective. Appendix A shows the XML configuration file used for this evaluation.

Deployment and initialization

A series of actions and interactions are expected to take place after the trader starts its system. They include shipping the trading computation to the Trading Computations Host,

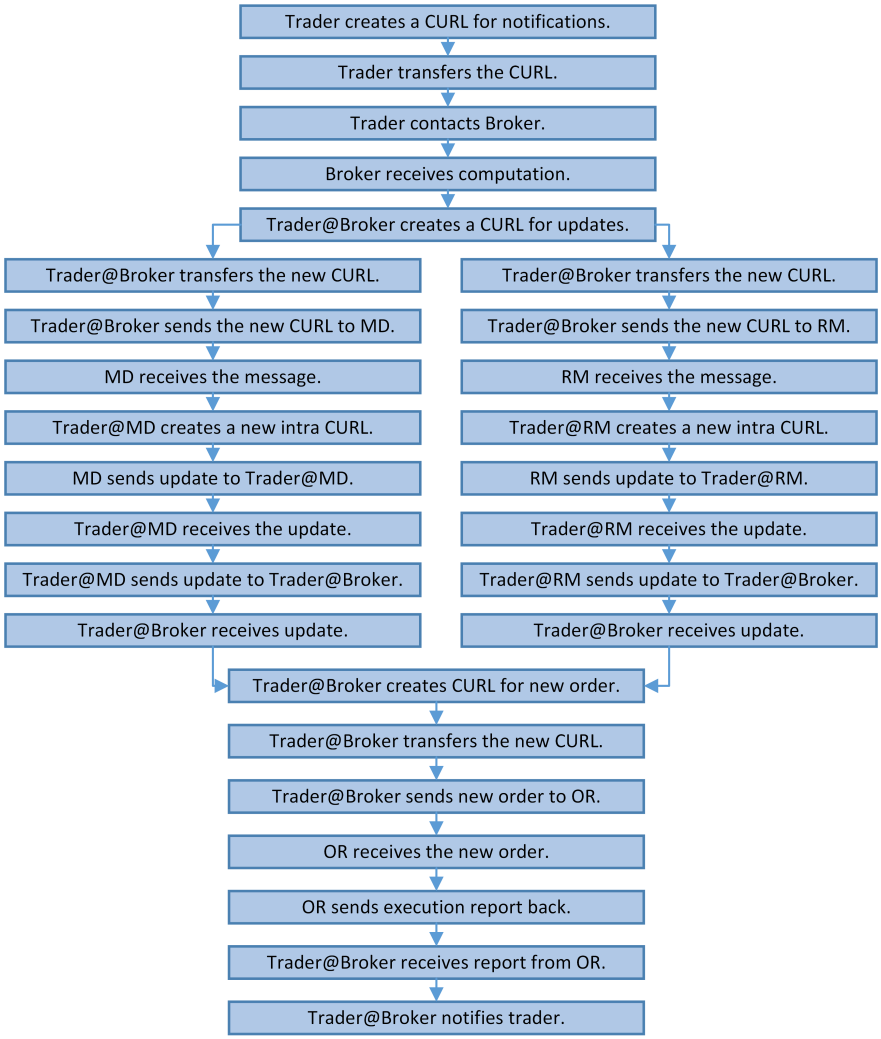

Figure 7.5: Verification model for deployment and initialization.

transferring a CURL for sending notifications back to the trader, registration in the Market Data and Risk Management servers, placing orders on the Order Router, and receiving execution reports. If any of these actions does not take place, we assume that the trading computation was not properly deployed and initialized. A diagram representing the execution model for deployment and initialization is included in Figure 7.5.

COMET verifies that all the anticipated capability events appeared, and that they fol-

lowed the expected order. After analysis, it produces an output stating the result (*PASS, FAILED, WARNING, ERROR, UNKNOWN*) of each evaluation.

Trading Computation opens a backdoor

In the decentralized system represented by the created prototype, the Trading Computation can only communicate back with the trader to send notifications. Any other form of communication with the trader is forbidden. Therefore, it is expected that the trader only exploits the CURL created by the Broker for the deployment service. If the computation running in the Broker creates a CURL, and it is exploited by the trader, then COMET will show that the verification model failed. Figure 7.6 shows the model for this verification.

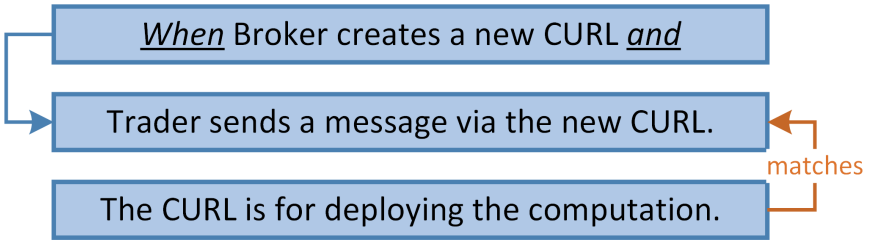

Figure 7.6: Verification model for detecting if a trading computation opens a backdoor.

For example, if the trader exploits a CURL created in the Broker, COMET will highlight it.

Number of orders placed

Some application-level and business-level evaluations are based on the number of orders placed by the trading computation. COMET supports verifications based on volume, that is, number of events within a time period. Based on the trading volume and the trading strategy, it is expected that the *base* trading computation places between 1 and 20 orders within 10 seconds. If, at any time, this requirement is not satisfied, COMET will highlight the time in which the volume exceeded or did not reach the expected number. Figure 7.7 shows the verification model for this requirement.

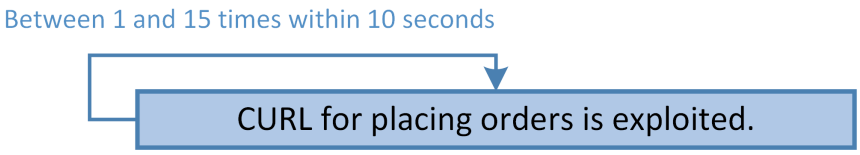

Figure 7.7: Verification model for number of placed orders.

7.4 Execution of *Base* Trading Computation

The *base* trading computation passes all the assertions in the verification model. It is used as a baseline for this evaluation. Table 7.3 shows a summary of the execution.

Duration:	5m 41s
Risk events:	101
Market events:	394
Orders:	260
Execution Reports:	787
Starts:	5
News:	796
Transfers:	658
Sends:	3487
Receives:	3487
Total capability events:	8433

Table 7.3: Summary of the execution of the base trading computation.

The following output is produced by COMET when it verifies the execution of this computation:

```
[PASS] Execution. Type: Unordered Message: N/A
    [PASS] Trading Computation is properly deployed and up & running. Type: Sequence
    Message: N/A
        [PASS] Trader provides a CURL for notifications. Type: Event Message: N/A
        [PASS] Trader includes its notification CURL. Type: Event Message: N/A
        [PASS] Trader ships a trading computation to the robot-server. Type: Event
        Message: N/A
        [PASS] robot-server receives the trading computation. Type: Event Message: N/A
        [PASS] Registration @ Market Data and Risk servers. Type: Exists Message: N/A
            [PASS] Trading Computation @ robot-server creates a new CURL to be sent to
            Risk and Market Data servers. Type: Event Message: N/A
```

[PASS] Registration @ Market Data and Risk servers. Type: Unordered Message: N/A

[PASS] Registration @ Market Data Server. Type: Sequence Message: N/A

[PASS] Trading Computation @ robot-server transfers the new CURL (to Market Data Server). Type: Event Message: N/A

[PASS] Trading Computation @ robot-server sends the new CURL to the Market Data server. Type: Event Message: N/A

[PASS] Market Data server receives the new CURL. Type: Event Message: N/A

[PASS] Trader @ Market Data creates a new CURL. Type: Event Message: N/A

[PASS] Market Data sends local update. Type: Event Message: N/A

[PASS] Trader @ Market Data receives update. Type: Event Message: N/A

[PASS] Trader @ Market Data sends update. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server receives update from Market Data Server. Type: Event Message: N/A

[PASS] Registration @ Risk Server. Type: Sequence Message: N/A

[PASS] Trading Computation @ robot-server transfers the new CURL (to Risk Server). Type: Event Message: N/A

[PASS] Trading Computation @ robot-server sends the new CURL to the Risk Management server. Type: Event Message: N/A

[PASS] Risk Management Server receives the new CURL. Type: Event Message: N/A

[PASS] Trader @ Risk Management creates a new CURL. Type: Event Message: N/A

[PASS] Risk Management sends local update. Type: Event Message: N/A

[PASS] Trader @ Risk Management receives update. Type: Event Message: N/A

[PASS] Trader @ Risk Management sends update. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server receives update from Risk Server. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server creates a CURL for an order. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server transfers the CURL to the Order Router. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server sends a message to the Order Router. Type: Event Message: N/A

[PASS] Order Router receives the order. Type: Event Message: N/A

[PASS] Order Router sends an execution report for the placed order to the Trading Computation @ robot-server. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server receives the execution report sent by the Order Router. Type: Event Message: N/A

[PASS] Trading Computation @ robot-server sends notification back to Trader. Type: Event Message: N/A

```
[PASS] Trading computation does not open a backdoor. Type: When Message: N/A
    [FAILED] Computation @ Robot Server creates a CURL. Type: Event Message: Event
    [place:inter (EQ), type:curl-new (EQ), source-island:robot-server (EQ),
    mq-time:1439840742418 (GE)] could not be found.
    [FAILED] Trader sends a message to the created CURL. Type: Event Message: Event
    [type:curl-send (EQ), source-island:trader (EQ),
    curl-id:a6e7c6bb-72fb-44d9-a5fc-5dc066542b6f (EQ), mq-time:1439840742417 (GE)]
    could not be found.
    [UNKNOWN] CURL matches deployer. Type: Match Message: N/A
[PASS] Number of orders is within expected values. Type: Volume Message: N/A
```

7.5 Problem-based evaluation

This section explains a set of problems that can arise in the proposed prototype during the execution of a trading computation. If none of these problems arise, then COMET will show that the entire verification passed. However, if any of the verifications does not pass, COMET will provide contextual information to the failure such as the expected capability event that was not found.

7.5.1 Trading Computation does not place orders

The Problem

From a debugging perspective, it is very difficult to localize the fault causing this problem. The computation placing orders depends on the right execution of the previous steps. For example, if the trading computation could not be deployed properly, or if it was unable to register on the Market Data Server, the computation will either not work, or will behave erratically. It is worth noting that the computation might still be able to place orders but, if it was not properly initialized, it is likely that it will not place the proper orders, which could be even worse that not placing orders at all.

Six different scenarios were created where distinct faults were introduced to manifest this problem. Below, each scenario is explained separately and details about its execution are

provided.

In order to deploy a computation in the Broker, the trader is expected to send a *thunk*[6] implementing the algorithm. In this scenario, the trader does send a function whose signature expects no parameters, however, the function body assumes that parameters exist. Consequently, when the function is evaluated, the execution engine will find variables out of lexical scope, and will terminate the execution immediately. Table 7.4 shows a summary of the execution.

Scenario 1.1: Computation is not properly initialized

Duration:	0m 20s
Risk events:	0
Market events:	0
Orders:	0
Execution Reports:	0
Starts:	5
News:	7
Transfers:	1
Sends:	2
Receives:	2
Total capability events:	17

Table 7.4: Summary of the execution of scenario 1.1.

When COMET evaluates the capability events collected when this scenario was run, it displays that the root evaluation failed:

```
[FAILED] Execution. Type: Unordered Message: N/A
```

But COMET also displays a breakdown of the execution. The following results help understand the reasons for this failure:

[6] A computation with no parameters.

```
(...)
[FAILED] Trading Computation @ robot-server transfers the new CURL (to Market Data
Server). Type: Event Message: Event [place:inter (EQ), type:curl-transfer (EQ),
source-island:robot-server (EQ), curl-id:39445b2f-79c2-4d7d-9faa-1e526a30de61 (EQ),
source-islet:trader (EQ), mq-time:1438986931136 (GE)] could not be found.
(...)
[FAILED] Trading Computation @ robot-server transfers the new CURL (to Risk Server).
Type: Event Message: Event [place:inter (EQ), type:curl-transfer (EQ),
source-island:robot-server (EQ), curl-id:39445b2f-79c2-4d7d-9faa-1e526a30de61 (EQ),
source-islet:trader (EQ), mq-time:1438986931136 (GE)] could not be found.
(...)
```

The results displayed above show the specific capability events that are missing in the verification. As a result, people responsible for the development and operation of the software can check the pieces of code and the deployment process responsible for carrying out those actions.

Scenario 1.2: Computation does not get deployed

In this scenario, instead of sending a thunk implementing the trading computation, the trader mistakenly sends a symbol with the function name[7]. Table 7.5 shows a summary of the execution.

[7]This error may seem farfetched but, for example, in Racket, the difference between sending a function and sending its name is only a single quote (e.g. trading-algorithm and 'trading-algorithm).

Duration:	0m 14s
Risk events:	0
Market events:	0
Orders:	0
Execution Reports:	0
Starts:	5
News:	6
Transfers:	1
Sends:	2
Receives:	2
Total capability events:	16

Table 7.5: Summary of the execution of scenario 1.2.

The breakdown showed by COMET when the evaluation fails includes the following message:

```
(...)
[FAILED] Trading Computation @ robot-server creates a new CURL to be sent to Risk and
Market Data servers. Type: Event Message: Event [place:inter (EQ), type:curl-new (EQ),
source-island:robot-server (EQ), source-islet:trader (EQ), mq-time:1438987974289 (GE)]
could not be found.
(...)
```

Since the computation could not be properly deployed, it is unable to perform its first required action (i.e. creating a CURL for the registration in the Market Data and Risk Management servers).

Scenario 1.3: Registration on the Market Data Server fails

In this scenario, the trading computation running in the Broker does not perform the registration on the Market Data Server properly. It does send the registration computation to the server but, when executed at the Market Data Server side, it does not use the proper API. Table 7.6 shows a summary of the execution.

Duration:	0m 56s
Risk events:	12
Market events:	0
Orders:	0
Execution Reports:	0
Starts:	5
News:	8
Transfers:	4
Sends:	30
Receives:	30
Total capability events:	77

Table 7.6: Summary of the execution of scenario 1.3.

The breakdown showed by COMET when the evaluation fails includes the following messages:

```
(...)
[FAILED] Market Data sends local update. Type: Event Message: Event [place:intra (EQ),
type:curl-send (EQ), source-island:market-server (EQ),
curl-id:384c6ef7-fa8a-4693-9209-b1490c67a9c6 (EQ), source-islet:updater (EQ),
mq-time:1439068347580 (GE)] could not be found.
(...)
```

Since the computation did not use the API offered by the Market Data Server properly, the server is unable to send updates.

Scenario 1.4: Registration on the Risk Management Server fails

In this scenario, the trading computation running in the Broker does not send the registration computation to the Risk Management Server, thus, it is unable to receive any risk update. Table 7.7 shows a summary of the execution.

Duration:	5m 31s
Risk events:	0
Market events:	391
Orders:	258
Execution Reports:	782
Starts:	5
News:	789
Transfers:	652
Sends:	3258
Receives:	3258
Total capability events:	7962

Table 7.7: Summary of the execution of scenario 1.4.

The breakdown showed by COMET when the evaluation fails includes the following message:

```
(...)
[FAILED] Trading Computation @ robot-server sends the new CURL to the Risk Management
server. Type: Event Message: Event [place:inter (EQ), type:curl-send (EQ),
source-island:robot-server (EQ), curl-id:6264c494-b240-43dc-892f-10c7dfbea378 (EQ),
source-islet:trader (EQ), mq-time:1439067709640 (GE)] could not be found.
(...)
```

Since the computation did not exploit the registration service's CURL, COMET found that the *curl-send* capability event was missing.

Scenario 1.5: Order Router is not working

In this scenario, the *base* trading computation is used, however, the Order Router is not working, thus, no orders can be placed. Table 7.8 shows a summary of the execution.

Duration:	6m 17s
Risk events:	98
Market events:	389
Orders:	257
Execution Reports:	0
Starts:	5
News:	786
Transfers:	649
Sends:	1638
Receives:	1368
Total capability events:	4446

Table 7.8: Summary of the execution of scenario 1.5.

The breakdown showed by COMET when the evaluation fails includes the following message:

```
(...)
[FAILED] Order Router receives the order. Type: Event Message: Event [type:curl-receive
(EQ), source-island:order-router (EQ), curl-id:feb0741f-4b42-4497-9c46-6205a97fe400 (EQ),
source-islet:order-receiver (EQ), mq-time:1439069720523 (GE)] could not be found.
(...)
```

Since the Order Router was not working, it was unable to receive the order from the trading computation.

Scenario 1.6: Algorithmic error in trading computation

In this scenario, the interaction with other systems work as expected. However, because of an algorithmic error in the trading computation, it is unable to place orders. Table 7.9 shows a summary of the execution.

Duration:	6m 38s
Risk events:	120
Market events:	390
Orders:	0
Execution Reports:	0
Starts:	5
News:	8
Transfers:	4
Sends:	1025
Receives:	1025
Total capability events:	2067

Table 7.9: Summary of the execution of scenario 1.6.

The breakdown showed by COMET when the evaluation fails includes the following message:

```
(...)
[FAILED] Trading Computation @ robot-server creates a CURL for an order. Type: Event
Message: Event [place:inter (EQ), type:curl-new (EQ), source-island:robot-server (EQ),
mq-time:1439070416796 (GE)] could not be found.
(...)
```

Since an algorithmic error prevented the computation from placing orders, COMET was unable to find the first of the capability events involved in placing orders.

7.5.2 Trading Computation does not send notifications to the trader

The Problem

For each order placed and each execution report received, the trading computation must send a notification to the trader. If that does not happen, the trader will not have visibility of the actions performed by the trading computation.

Scenario 2.1: Computation does not exploit its CURL for notifications

The entire execution of the trading computation works as expected with the exception of sending notifications back to the trader. Table 7.10 shows a summary of the execution.

Duration:	8m 31s
Risk events:	154
Market events:	389
Orders:	257
Execution Reports:	780
Starts:	5
News:	786
Transfers:	649
Sends:	2517
Receives:	2517
Total capability events:	6474

Table 7.10: Summary of the execution of scenario 2.1.

The breakdown showed by COMET when the evaluation fails includes the following message:

```
(...)
[FAILED] Trading Computation @ robot-server sends notification back to Trader. Type:
Event Message: Event [place:inter (EQ), type:curl-send (EQ), source-island:robot-server
(EQ), curl-id:7ae2fea9-7214-4daf-9e9f-a0d02fcdc043 (EQ), mq-time:1439068565060 (GE)]
could not be found.
(...)
```

Since the CURL for sending notifications back was not exploited, COMET detects the missing capability and highlights the problem.

7.5.3 Trading Computation opens a backdoor

The Problem

A safe (i.e. not under attack) host may be affected by an evil computation if the host is poorly protected or if, under the impression that the computation origin is secure, the host grants privileges to the computation that allows it to communicate back with a malicious host.

In COAST, all communications are enabled and constrained by Capability URLs. As a result, if a computation needs to receive a message, it has to create a CURL and transfer it to whomever is interested in communicating with that computation.

Monitoring the CURLs created by a visiting computation can prevent malicious external hosts from establishing communications with the computation.

In the prototype, the trader is only allowed to talk to the Broker via its deployment service CURL. If the trader sends a message to its trading computation or to any other computation running in the Broker, it will be detected by COMET and corrective actions can be taken such as revoking the CURL or killing the computation.

Scenario 3.1: Computation opens a backdoor and trader exploits it

After the trading computation is deployed, it creates a CURL to enable incoming communications and sends it back to the trader via its notifications CURL. Table 7.11 shows a summary of the execution.

Duration:	7m
Risk events:	112
Market events:	389
Orders:	257
Execution Reports:	780
Starts:	5
News:	782
Transfers:	646
Sends:	3468
Receives:	3468
Total capability events:	8369

Table 7.11: Summary of the execution of scenario 3.1.

COMET detects that a CURL created in the Broker was exploited by the trader, but the CURL was not the one for the deployment service:

```
(...)
    [FAILED] Trading computation does not open a backdoor. Type: When Message: curl-id is
    different: e95d56cd-7a48-4344-af91-55a984ade55a
        [PASS] Computation @ Robot Server creates a CURL. Type: Event Message: N/A
        [PASS] Trader sends a message to the created CURL. Type: Event Message: N/A
        [FAILED] CURL matches deployer. Type: Match Message: curl-id is different:
        e95d56cd-7a48-4344-af91-55a984ade55a
(...)
```

7.5.4 Trader uses a different service level

The Problem

The speed of market data delivery, decision making and order execution is tremendously important in electronic trading. It is estimated that 1-millisecond advantage is worth one million dollar a year to a major brokerage firm [31].

Brokerage firms offer different levels of service at different prices. Each plan may include direct connections to exchanges, lower latency, more processing power, and other features.

COAST CURLs and execution sites enable providers to offer highly customizable services to clients. For example, a service provider can offer execution hosts running on hardware with different specifications, and the customer can contract the hardware specifications that better fit its business or trading strategy. CURLs can be used to constrain who can deploy trading computations in an execution site, and the circumstances in which it can be done. Additionally, custom-tailored APIs can be directly associated to CURLs so that service providers can determine which functions and implementations are available to each client.

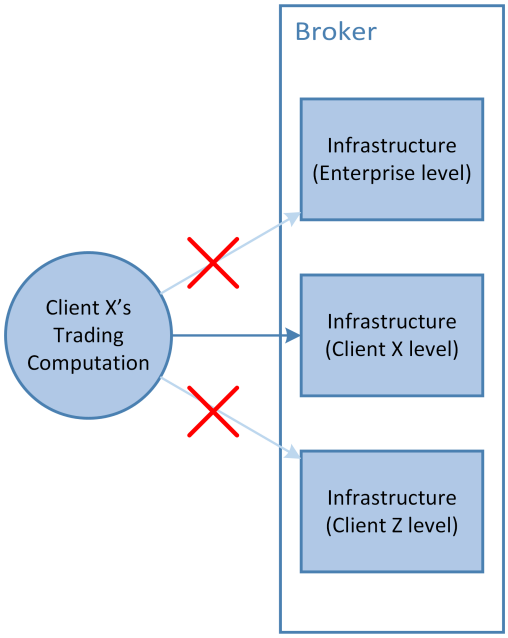

Figure 7.8: The trading computation must use the CURL corresponding to the contracted service level.

Using COMET and capability accounting, providers have an extra layer of verification to make sure that they are offering the right service level to each customer. Suppose Client X's trading computation must be deployed on an execution host customized for it as shown in Figure 7.8 ("Broker (Client X level)"). The service provider can create a CURL that points to the required execution host, and includes the functions and implementations negotiated.

Scenario 4.1: Computation uses a CURL for a different service level

In this scenario, the Broker offers a Premium service in addition to the basic service the trader contracted. Somehow, the trader got the CURL of the Premium service and deployed its computation via that CURL. Table 7.12 shows a summary of the execution.

Duration:	5m 54s
Risk events:	102
Market events:	389
Orders:	257
Execution Reports:	779
Starts:	5
News:	786
Transfers:	650
Sends:	3450
Receives:	3450
Total capability events:	8369

Table 7.12: Summary of the execution of scenario 4.1.

COMET detects that the trader used a CURL for deployment different from the one it is supposed to use:

```
(...)
[FAILED] Trader ships a trading computation to the robot-server. Type: Event Message:
Event [place:inter (EQ), type:curl-send (EQ), source-island:trader (EQ),
curl-id:b927dace-e6b5-4ca7-9c22-b1d4e3ba4b9f (EQ), mq-time:1439066901339 (GE)] could not
be found.
(...)
```

Because of the way the verification model was created, COMET does show that an assertion failed, although the message may be misleading. It highlights that there was a problem when deploying the computation but does not clarify exactly what the problem was. Modifying the model to assert whether subsequent system actions were taken would help providing clearer information.

7.5.5 Trading computation places too many or too few orders

The Problem

Trading algorithms work with very large amounts of data. Computations automatically dispatch, based on market conditions, several buy and sell orders, and cancel and replace some of them within a few seconds. Although the number of orders placed may vary considerably, depending on external conditions and the trading strategy, there are limits that the Trader can foresee. If the number of orders goes beyond those limits, it is very likely that the trading computation is not working as expected.

Scenario 5.1: A fault in the trading computation causes it to place more orders than expected

The algorithm that places orders has a fault that causes that multiple orders are placed for each one intended. Table 7.13 shows a summary of the execution.

Duration:	10m 33s
Risk events:	195
Market events:	389
Orders:	3062
Execution Reports:	741
Starts:	5
News:	786
Transfers:	3455
Sends:	9161
Receives:	6349
Total capability events:	19756

Table 7.13: Summary of the execution of scenario 5.1.

The COMET Volume evaluation detects that the number of orders exceeded the limit and displays the following message:

```
(...)
[FAILED] Number of orders is within expected values. Type: Volume Message: Max volume was
exceeded. Seen: 21 at 1439083374352. Expected:[1,20]
(...)
```

Scenario 5.2: A fault in the trading computation causes it to place less orders than expected

The algorithm that places orders has a fault that causes only some of the intended orders are actually placed. Table 7.14 shows a summary of the execution.

Duration:	5m 45s
Risk events:	99
Market events:	389
Orders:	60
Execution Reports:	189
Starts:	5
News:	170
Transfers:	145
Sends:	1559
Receives:	1559
Total capability events:	3438

Table 7.14: Summary of the execution of scenario 5.2.

The COMET Volume evaluation detects that the number of orders does not reach the lower limit and displays the following message:

```
(...)
[FAILED] Number of orders is within expected values. Type: Volume Message: Min volume was
not reached. More than 10 seconds passed with no events at 1439073236442. Expected:[1,20]
(...)
```

7.6 Summary

This Chapter shows how COMET, a tool based on capability accounting, was used to verify the proper execution of a COAST-based electronic trading system prototype. The intention was to assess that capability accounting is a suitable technique to help developers and operators to design, develop, operate, and maintain secure decentralized systems. A verification model was created, and used in COMET to verify the execution of different trading computations. The *base* trading computation, free of any sign of failure, passed the evaluation with no problems. Looking at the output produced by COMET, it can be seen that all verifications passed. Then, the same evaluations were performed on faulty computations. COMET was able to detect the presence of an unexpected behavior in all of them, and to highlight which capability event was missing in all of them, except for one but a modification in the verification model would solve it. Highlighting the problematic capability event would help developers localize where the fault is.

Chapter 8

Discussion

In Chapter 1, the research problem and research questions are presented, that is, the challenges of designing, developing, operating, and maintaining decentralized systems, and how capability accounting can be used to help address or reduce the risk and impact of those problems. This work explains what a capability is, how capabilities are abstracted and represented in the COAST architectural style, and it revisits the COAST principles and the Motile/Island reference implementation. Chapter 6 explains why electronic trading systems are decentralized systems, and discusses the relevancy of software in the financial industry, and the importance of the financial industry in the global economy. In order to address the research questions, a prototype of a COAST-based trading system, and a monitoring system were built; several trading computations were created to run in the prototype to show that capability accounting can be used to verify the proper execution of software, help developers with fault-localization, control the compliance with business rules, and monitor security.

The analysis of capability accounting was based on the use of the COAST architectural style. Implementing capability accounting required the instrumentation of the Motile/Island platform to capture the creation, exploitation, and transfer of Capability URLs. Those capability events were then stored in a database for future inspection. Finally, a verification model was created in COMET to compare expected capability patterns with observed

patterns.

This Chapter includes lessons learned throughout this research project, and certain topics that are worth presenting and discussing, although they were not fully explored.

8.1 Application-agnostic logging

As Section 6.2 explains, capturing run-time information for investigation or forensic analysis in high-availability, real-time systems can be extremely challenging:

- First, security and privacy concerns difficult or prevent direct connections to the applications under inspection. Because servers where infrastructure and trading systems run have a protected perimeter, it is not possible to interact with them from outside. Thus, information retrieval must be done in-site, depending on the availability of the IT staff to grant access, or through various levels of indirection, which makes the analysis much more troublesome.

- Second, generated log files are, many times, the only information that developers count with for investigating an ongoing issue, or to do forensic analysis. As a result, logging implementation becomes very intricate because too much logging will certainly penalize the application performance and consume too much storage, but little logging can put at risk the proper analysis. Consequently, application developers must spend a considerable time designing logging strategies, and carefully choosing what information is logged, how, and where.

- Third, some companies do not let vendors ship log files outside their execution environment because they might contain sensitive information that affects their customers. Admittedly, companies allowing vendors to get their log files are indeed exposing sensitive information because transactions could be posted in log files.

Consequently, an expected situation in most IT environments like inspecting the execution of software to investigate suspicious activity becomes tremendously complicated in the financial trading industry. The aforementioned issues have an impact on how systems are debugged and configured, and in the amount of time involved in those activities. Additionally, because logging must be carefully planned in advance, developers must spend extra time designing and implementing smart strategies to log information, but with a lack of certainty about whether the captured information will be enough or not when an issue arises.

Several parties involved in the design, development, operation, and maintenance of software will be benefited from the practice of capability accounting:

- Application developers can focus on logging higher-level information. Since capability accounting can be used to verify software execution at system-level and at application-level, developers can reduce the amount of information they need to log, focusing on business or input-dependent information. Admittedly, uncountable situations require custom application-level logging but this work proves that the proposed approach does work and can be further developed to increase the coverage of situations that can be improved by using capability accounting.

- System software developers can put in place streamlined and optimized methods for capturing capability events in order to minimize the performance penalization.

- Clients can share logs of capability events with vendors. Because those events do not contain sensitive information, clients are not at risk of disclosing private information, and vendors can use it to investigate erroneous or suspicious behavior on their applications.

8.2 Interpretation depends on the nature of available events

Dealing with decentralized systems requires a cautious examination of the integrity of the data used for analyses and the reliability of its source. Several questions need to be considered

in order to know whether a piece of data is safe and reliable or not:

- Where was the information produced? Is that a trusted source? By default, there are no trusted sources unless mechanisms in place can guarantee it. Furthermore, even peers that were reliable in the past may have been compromised, and affected in order to produce information that *looks* real but it is not. For example, the name of an islet is determined by the parent islet who creates it. Peers have little or no control over islets naming, and there is no way to have certainty about which islet has produced a message.

- Could someone have forged or tampered with the information? Cryptographic mechanisms must assure that information has not been compromised during its generation or transmission. For example, CURLs in COAST are unforgeable, tamper-proof structures that can be safely consulted because the underlying security infrastructure guarantees that its content is intact.

- How accurate is the information? Regardless of where a message is produced, some information is inherently inaccurate when distributed. For example, timestamping a message can occur when the message is created, during its transmission or when it is received. It can also be done in multiple occasions during its lifecycle. Nevertheless, when the source and the target of a message are different peers, unless one trusts another (which should not happen unless one peer can prove its reliability to the other), the organization who is analysing capabilities will use their own information. That would be, as an example, if peer a sends a message to peer b, and the organization responsible for peer b is performing an analysis, the organization will timestamp the message when it arrives at b, and will use only that information because peer a may have timestamped the message in a wrong manner to mislead peer b. Let us say that the message that arrives at b has to be correlated with another message coming from peer c. Because exact latency cannot be determined in advance, the order of arrival may

be different from the order in which messages were sent [11]. Analyzers must account for this kind of inaccuracies. It is worth noting that although clock synchronization is a problem that has already been solved, when analyzing security, information from an untrusted peer has to be taken as potentially compromised.

- Could a human mistake have altered or have produced erroneous information? Even in a world of perfect security, software is not completely autonomous; instead, it is designed, developed, maintained, and most of the times, operated by people. As a result, errors introduced by improper human actions are likely to happen. Synergetic interactions among components determine the systemic behavior of a software. This statement is true for all kind of software systems, but it is even more relevant in decentralized systems, where the layering of software, people and organizations affecting the overall system behavior is much more complex. Section 6.1.2 lists only a few from the total number of incidents that affected the financial trading industry, and mentions their causes and consequences. As it can be seen, many of them were caused by human mistakes or configuration issues.

- Do we count with all the required information? Capturing run-time information about the execution of a software system is not a trivial process. Data has to be acquired, transmitted, and stored for analysis, as explained in Chapter 3, but these activities may face challenges depending on the data volume. When large amounts of data have to be logged, the logging routines compete for CPU cycles with the applications they work for. Additionally, a synchronous logging implementation will penalize the application even more, but an asynchronous implementation will need to use internal buffers that could overflow, and may require more RAM memory. As a result, software developers must choose which information is going to be logged, and configure the application to include only the most relevant logging levels, excluding the most verbose ones. Furthermore, many times, developers have to implement smart logging strategies. As

a consequence of all these issues, when an investigation is performed, it is important to know the logging policies that were in place when the information was captured.

The previous questions have to be answered in order to decide whether a specific type of analysis can be performed, and which information is available for it. Less risky analyses such as fault localization can make use of data even if the data is not guaranteed to be reliable from a security perspective. A classical risk analysis can help us determine whether a piece of information can be used or not. The fact that a peer may have been compromised does not mean that it has been but there is some level of likelihood that it has. Depending on the severity of the analysis to perform, the likelihood of receiving forged data, and the impact of producing the wrong result will determine what information has to be ignored. A more formal representation of these concepts is as follows:

Let x be a piece of data that has been acquired, $p(x)$ the probability that x has been forged or tampered, $i(x)$ the impact of producing an erroneous result if x is used:

$R(x) = p(x) * i(x)$ where $R(x)$ is the risk associated to using x in an analysis.

This work is neither intended to go in depth on risk analysis nor to provide an accurate model for data selection, however, the previous formula can help us understand the principal variables involved in choosing the data for a given analysis. For example, when doing security analysis, the consequences of missing the exploitation of a vulnerability may be disastrous, thus $i(x)$ will be high, and even if $p(x)$ is low, the analysis may not be worth because of its associated risk. However, when debugging an error, the impact of inspecting a piece of code that does not contain a fault is much lower, meaning that a piece of data will be ignored only if the probability that it is not reliable ($p(x)$) is very high.

8.3 Analyzing domain-specific issues using architectural accountability

In this discussion, I propose a classification of software issues that does not pretend to be universally applicable but to guide the creation of scenarios where capability accounting is employed.

Although all of the issues analyzed in Chapter 7 are framed within the financial trading domain, many of them could be easily extrapolated to other domains. For example, preventing computations from opening a backdoor is desirable in electronic trading but also in any other domain. Furthermore, the assertion created in COMET could be used with no modifications (except for names and IDs) in other contexts. However, the computation deployed by a trading firm in the broker's host satisfying a contractual obligation (e.g. placing more than the minimum expected number of orders) is specific to the domain in which the application operates. Following this reasoning, issues can be catalogued as domain-independent and domain-dependent.

Domain-independent issues are inherently technical, and have generally to do with an application not behaving as the developers expected (i.e. a failure). Possible reasons for these effects are defects introduced by developers, the exploitation of security vulnerabilities, mistreatment of unexpected inputs (e.g. different data formats, erroneous implementations of software contracts or protocols), and the unexpected response of other systems in which our application depends on (i.e. the other system does not respond or provides a response that does not let our application to continue). In the created prototype, the deployed computation has to be deployed and started, it has to register in the Market Data and Risk Management servers and enable them to communicate back, place orders, and receive execution reports. Although the actions are carried out within the financial trading domain, it is clear that the described scenario could be abstracted and extrapolated to other domains; that is, the scenario could also be described as a computation x that has to be deployed, started, it has

to subscribe to peers A and B, and then establish a request-response communication pattern with peer C. It can be seen that the nature of the data is ignored because the evaluation only encompasses action and interaction patterns.

Domain-dependent issues can be also dissected into technical or business-related. A trading computation that does not place the right orders could be affected by a bug in the trading algorithm (i.e. technical cause), or by a flawed trading strategy (i.e. business-related cause). If a trader deploys a computation that behaves in a certain way with the only goal of misleading other computations and making profit out of it, that is an unfair practice that goes against trading regulations, thus, it should be prevented or, at least, detected.

My experience developing the prototype and the trading computations, and creating scenarios in COMET showed me that domain-independent issues are easier to verify using capability events than domain-dependent issues. The fact that communications between peers can be abstracted as interactions between components, regardless of their nature facilitates the analysis based on capability accounting.

Verifying the compliance with business rules requires to figure out a mapping between the rules and capability events. For example, the created prototype assumes that, in order to place an order, the trading computation creates a CURL, and transfers it via the Order Router CURL. Whereas this is true for the current implementation, it could also be different, depending on the APIs provided (i.e. the binding environments), and the preference of the software developers. Nevertheless, it is worth noting that it does seem to be the best design solution in COAST: creating a new CURL for each order enables more fine-grained tracking and control.

The resulting properties of a COAST-based software design are not part of this work, and certainly requires further investigation. In particular, two salient research questions are posed for future investigation:

1. What are the implications of designing a COAST-based software for capability accounting? In the same way testability has been deeply studied in the past [8, 33],

designing a piece of software with capability accounting in mind and its effects would also need to be studied. In particular, what trade-offs are found when considering accountability as a software property?

2. When building the prototype, some decisions were made in order to provide accountability. They *seemed*, anyways, to be better design decisions. Are best practices in COAST aligned with producing accountable software?

8.3.1 A domain-dependent case using capability accounting

On June 2012, Igor Oystacher, whom securities traders call "The Russian", sent thousands of buy and sell orders on the London Exchange. But he canceled many of those orders milliseconds after placing them, documentation show, in what the exchange alleges was part of a trading practice designed to trick other investors into buying and selling at artificially high or low prices. This bluffing technique is called "spoofing" and has long been used in trading. Hope [25] explains that a spoofer might deceive other traders into thinking oil prices are falling, say, by offering to sell futures contracts at $45.03 a barrel when the market price is $45.05. After other sellers join in with offers at that lower price, the spoofer quickly pivots, canceling his sell order and instead buying at the $45.03, price he set with the fake bid. The spoofer, who has now bought at two cents under the true market price, can later sell at a higher price, perhaps by spoofing again, pretending to place a buy order at $45.04 but selling instead after tricking rivals to follow. Repeated many times, spoofing can produce big profits.

Imagine a COAST-based Order Router system implemented as shown in Figure 8.2. The Order Router has a CURL for a trader to place new orders. Once a new order is placed, the Order Router creates a brand new CURL, whose lifespan will be the same as the order's lifespan, that the trader can use to cancel or replace the order. When the order is filled, the Order Router revokes the new CURL as it can no longer be exploited. In this system, it is easy to correlate transactions related to each individual order and, eventually, take preventive

or corrective actions such as defining throttles for messaging, or revoking the CURL.

The Chicago Stock Exchange[1] is offering SNAP™ [2], an auction order that is hold for less than a second but gives enough time to all the players for a fair trading [29]. Regarding frequent batch auctions, Budish [13] says that discrete time reduces the value of tiny speed advantages, and the auction transforms competition on speed into competition on price. Consequently, frequent batch auctions eliminate the mechanical arbitrage rents, enhance liquidity for investors, and stop the high-frequency trading arms race. COAST has native support for this type of orders by using gates to define the frequency in which a CURL can be exploited. Furthermore, if an external monitoring system such as a real-time version of COMET detects suspicious activity carried out by a trader, it could eventually change the order's CURL gate to restrict the frequency in which the order can be cancelled or replaced, thus, mitigating the effects of a spoofer. Instead, if a spoofing attempt is confirmed, the monitoring system could revoke the CURL associated to specific orders, preventing the trader from continuing the operation on those orders.

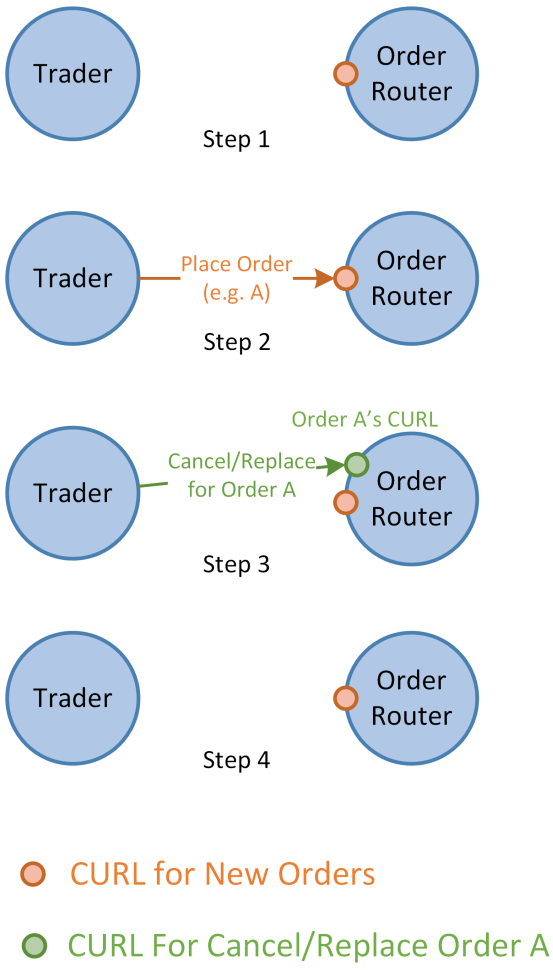

Figure 8.2: Suggested order router to handle spoofing.

[1] http://www.chx.com
[2] http://www.chx.com/snap/

8.3.2 Detecting and preventing spoofing automatically

Assuming that it is technically possible to monitor all the necessary orders in real-time, there is a major challenge in detecting illegal activities: uncovering the trader's intention. Because spoofing and other illegal strategies are based on valid market transactions, the scene has to be examined as a whole, including its context. The spoofing case explored in this section is based on quickly canceling orders right after they were placed. However, canceling an order is an absolutely valid action. In relation to the spoofing case Hopes [25] illustrates, "A trader might cancel after the market heads in an unexpected direction or when a news flash suggests a different trade is in order", and warns, "some traders say it can be difficult to distinguish between illegal market manipulation and savvy tactics to conceal the size of an intended trade – a technique used by traders since financial markets' earliest days."

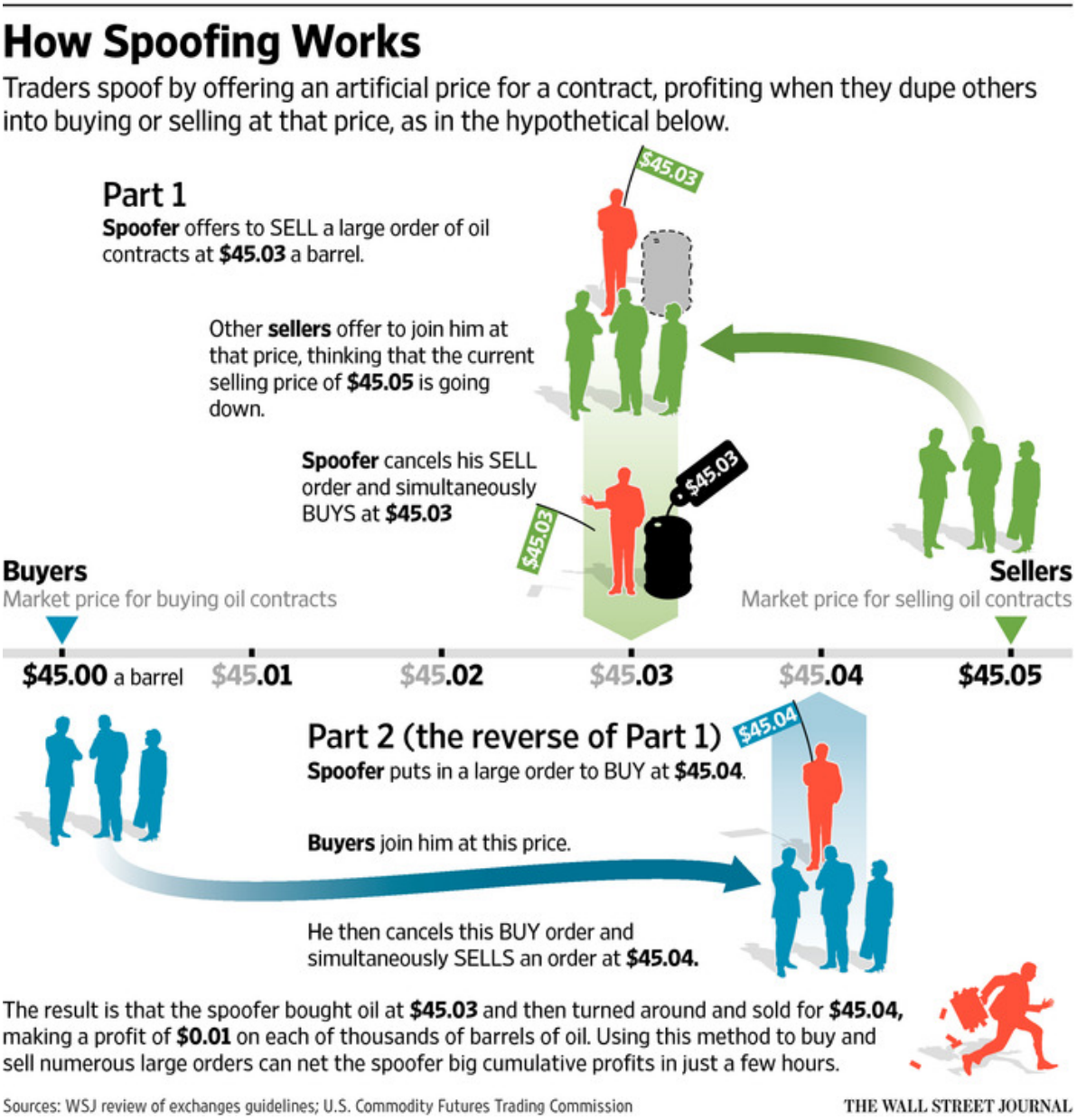

Figure 8.1: How spoofing works in electronic trading. [25]

Chapter 9

Conclusion

Throughout this work, I explored the challenges of designing, developing, operating, and maintaining secure decentralized systems, and assessed the feasibility of using architectural accountability to overcome or mitigate some of those difficulties. Chapter 1 explains what decentralized systems are, their characteristics, and the challenges associated to them. It also relates the nature of these systems with the COmputAtional State Transfer (COAST) architectural style, and explains why it is a good fit for such systems. Chapter 2 revisits COAST, its principles and main elements, and the Motile/Island reference implementation. Chapter 3 defines what capability accounting is, its purpose, structural elements and suggests a framework for monitoring including: capture, storage, query, data preparation, evaluation, reporting, and action, and what can or must be done during each activity. Chapter 5 presents COast Monitoring Event Tool (COMET), a tool that enables the verification of software execution by correlating and analyzing capability events captured by an instrumented version of Motile/Island, and comparing them with expected patterns. This chapter also presents the design of COMET, and the technology used to build it. Chapter 6 presents the relevance of financial trading in the global economy, and explains why software systems in that domain must be considered as decentralized systems. It also presents challenges in the financial trading industry, and explains how some activities such as debugging and troubleshooting

have to be carried out because of security and privacy concerns, and the constraints about how software systems must be operated. The chapter includes as well impactful cases from the last five years that still resonate in the trading circles and costed several hundred million dollars. Chapter 7 presents the evaluation performed for this work to assess whether it is feasible to use capability accounting to help developers and system operators. It describes the created prototype and trading computations, the implemented trading strategy, and a list of issues that were solved using capability accounting and COMET, and details the way in which the issues were approached. Chapter 8 includes a discussion about various topics related to capability accounting. In particular, it explores the relationship between application logging and capability accounting, introduces a risk-based treatment of data for capability analysis, and describes the difference between domain-independent and domain-dependent issues, and the challenges associated to the latter. It concludes by exposing a spoofing case occurred in 2012, and how a COAST-based system could account for the required information to help solve such case. It illustrates a prototype of a COAST-based Order Router, whose design was prepared taking capability accounting into consideration, and embraces mitigation and prevention actions. Finally, a discussion about the challenges of uncovering actual spoofers' intentions is included, highlighting the difficulty of detecting whether an action should be considered as valid or if it is part of a strategy to make profit illegally.

The contributions and outcomes of this research work are, therefore:

- The assessment of a novel technique for building and operating secure decentralized systems based on architectural accountability: capability accounting. I demonstrated that tracking the creation, exploitation, and transfer of Capability URLs, a COAST fundamental element, several issues present in decentralized systems can be mitigated or prevented.

- An evaluation of capability accounting within the financial trading domain. A prototype of an electronic trading system, where systems presumably developed by different

organizations interact to let individuals and companies trade securities. The execution and operation of various scenarios were verified using COast Monitoring Event Tool (COMET).

- A number of techniques and operators to represent and analyze capability events: *Sequence*, *Volume*, *Match*, *When*, *Exists*, *Unordered*, *Not*, and *Or*.
- Evidence that strongly suggests that capability accounting can be used for domain-dependent and domain-independent issues, at technical and business levels.

In particular, interesting findings related to COAST were obtained:

- COAST, as an architectural style, strongly supports architectural accountability.
- COAST is very well-suited for capability accounting.
- Capability accounting can be used to obtain interesting information about COAST systems.

The main research questions I embarked to answer in this work are:

- **Is capability accounting a well-suited form of architectural accountability?** As a result of this study, I demonstrated that capability accounting as a form of measurement enables architectural accountability. It is possible to obtain insightful information such as behavior, effects, and properties about software systems.
- **How can capability accounting be used to help build, operate and maintain secure decentralized systems?** I proposed a framework that covers the capture, transfer, storage, query, data preparation, evaluation, reporting, and action of capability events. An instrumented version of the Motile/Island implementation captures and ships out capability events to be stored in a database. These events are then inspected and correlated by COMET, alerting when the observed event patterns differ from the expected event patterns. Finally, decisions can be made upon the analysis results.

- **Is the financial trading domain a suitable domain for capability accounting?** Through the creation of a prototype of an electronic trading system, and trading computations that operate in that system, running in the instrumented version of Motile/Island, generating capability events that were later on correlated and analyzed by COMET, a tool especially created for this research work, I demonstrated that the financial trading domain is a suitable domain for capability accounting. Furthermore, the evaluation results strongly suggest that the proposed technique works with domain-independent issues, and that it can also be applied to issues in other domains, as long as the mapping between system actions and produced capability events is clear and well-defined.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1449159 and by Bloomberg L.P. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or Bloomberg L.P.

Bibliography

[1] STOMP Protocol Specification, Version 1.1. https://stomp.github.io/stomp-specification-1.1.html. [Online; accessed 10-August-2015].

[2] Direct Market Access. http://www.londonstockexchange.com/prices-and-markets/stocks/tools-and-services/direct-market-access/direct-market-access.htm, 2015. [Online; accessed 01-August-2015].

[3] Exchange. http://www.investopedia.com/terms/e/exchange.asp, 2015. [Online; accessed 01-August-2015].

[4] Trader. http://www.investopedia.com/terms/t/trader.asp, 2015. [Online; accessed 01-August-2015].

[5] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[6] P. Bajpai. Strategies And Secrets Of High Frequency Trading (HFT) Firms. http://www.investopedia.com/articles/active-trading/092114/strategies-and-secrets-high-frequency-trading-hft-firms.asp, 2015. [Online; accessed 01-August-2015].

[7] J. T. Behrens. Principles and procedures of exploratory data analysis. *Psychological Methods*, 2(2):131, 1997.

[8] R. V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[9] I. Board. Ieee standard classification for software anomalies. *IEEE Std*, 1044, 1993.

[10] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The keykos nanokernel architecture. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 95–112, 1992.

[11] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[12] S. Brush, T. Schoenberg, and S. Ring. How a Mystery Trader With an Algorithm May Have Caused the Flash Crash. http://www.bloomberg.com/news/articles/2015-04-22/mystery-trader-armed-with-algorithms-rewrites-flash-crash-story, 2015. [Online; accessed 01-August-2015].

[13] E. B. Budish, P. Cramton, and J. J. Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *Fama-Miller Working Paper*, pages 14–03, 2013.

[14] F. T. Community. What is FIX? http://www.fixtradingcommunity.org/pg/main/what-is-fix, 2015. [Online; accessed 01-August-2015].

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, Mar. 1966.

[17] A. S. Engineering. Argo Messaging System? http://www.argocons.com/rmcast.html, 2015. [Online; accessed 01-August-2015].

[18] R. Finberg. Global FX Average Daily Volumes in 2014 to Reach $5.5 Trillion: Reporte. http://www.financemagnates.com/forex/analysis/aite-group-expects-global-fx-average-daily-volumes-2014-reach-5-5-trillion/, 2014. [Online; accessed 01-August-2015].

[19] I. M. Found. World Economic Outlook Database. http://www.imf.org/external/pubs/ft/weo/2015/01/weodata/weorept.aspx?pr.x=33&pr.y=7&sy=2014&ey=2015&scsm=1&ssd=1&sort=country&ds=.&br=1&c=111&s=NGDPD%2CNGDPDPC%2CPPPGDP%2CPPPPC&grp=0&a=, 2015. [Online; accessed 01-August-2015].

[20] P. Gillespie. Trading resumes on NYSE after nearly 4-hour outage. http://money.cnn.com/2015/07/08/investing/nyse-suspends-trading/, 2015. [Online; accessed 01-August-2015].

[21] M. M. Gorlick, K. Strasser, and R. N. Taylor. Coast: An architectural style for decentralized on-demand tailored services. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 71–80. IEEE, 2012.

[22] M. M. Gorlick and R. N. Taylor. Motile: Reflecting an architectural style in a mobile code language. 2013.

[23] M. M. Gorlick and R. N. Taylor. Communication and capability urls in coast-based decentralized services. In *REST: Advanced Research Topics and Practical Applications*, pages 9–25. Springer, 2014.

[24] C. Group. Multi-Protocol Support. http://www.greenlinetech.com/support/multiprotocol.php, 2014. [Online; accessed 01-August-2015].

[25] B. Hope. As 'Spoof' Trading Persists, Regulators Clamp Down. http://www.wsj.com/articles/how-spoofing-traders-dupe-markets-1424662202, 2015. [Online; accessed 01-August-2015].

[26] A. Jeffery. Goldman trading glitch could cost more than $100 million. http://www.cnbc.com/id/100976404, 2013. [Online; accessed 01-August-2015].

[27] M. Kaminsky and E. Banks. Sfs-http: Securing the web with self-certifying urls. Technical report, Citeseer, 1999.

[28] T. C. Lin. New investor, the. *UCLA L. Rev.*, 60:678, 2012.

[29] L. Marek. The Chicago Stock Exchange's big idea to slow down trading. http://www.chicagobusiness.com/article/20150513/NEWS01/150519894/the-chicago-stock-exchanges-big-idea-to-slow-down-trading, 2015. [Online; accessed 01-August-2015].

[30] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 105–116, New York, NY, USA, 2010. ACM.

[31] R. Martin. Wall Street's Quest To Process Data At The Speed Of Light. http://www.informationweek.com/wall-streets-quest-to-process-data-at-the-speed-of-light/d/d-id/1054287?, 2007. [Online; accessed 01-August-2015].

[32] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, Jan 2000.

[33] S. Mouchawrab, L. C. Briand, and Y. Labiche. A measurement framework for object-oriented software testability. *Information and software technology*, 47(15):979–997, 2005.

[34] NASDAQ. Ouch. http://www.nasdaqtrader.com/Trader.aspx?id=ouch, 2015. [Online; accessed 01-August-2015].

[35] NYSE [nyse], July 2015. (1 of 3) The issue we are experiencing is an internal technical issue and is not the result of a cyber breach. [Tweet]. Retrieved from https://twitter.com/nyse/status/618818929906085888.

[36] W. F. of Exchanges. Monthly reports. http://www.world-exchanges.org/statistics/monthly-reports, 2015. [Online; accessed 01-August-2015].

[37] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[38] Oxford Dictionaries. Event. http://www.oxforddictionaries.com/us/definition/american_english/event, 2015. [Online; accessed 16-August-2015].

[39] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[40] B. V. Protopopov and A. Skjellum. A multithreaded message passing interface (mpi) architecture: Performance and program issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, 2001.

[41] J. H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 17(7):388–402, July 1974.

[42] M. Schapiro. Testimony Concerning the Severe Market Disruption on May 6, 2010. https://www.sec.gov/news/testimony/2010/ts051110mls.htm, 2015. [Online; accessed 1-August-2015].

[43] S. Seth. Basics of Algorithmic Trading: Concepts and Examples. http://www.investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp, 2015. [Online; accessed 01-August-2015].

[44] I. Staff. The Stop-Loss Order - Make Sure You Use It . http://www.investopedia.com/articles/stocks/09/use-stop-loss.asp, 2015. [Online; accessed 01-August-2015].

[45] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

Appendices

A COMET Configuration

This appendix includes the XML content of the COMET configuration file used in Section 7.3.1. The CURLs are represented using their IDs, they correspond with the following components:

- **b927dace-e6b5-4ca7-9c22-b1d4e3ba4b9f:** Service to deploy computations at the Trading Computations Host.
- **6264c494-b240-43dc-892f-10c7dfbea378:** Service to subscribe for events in the Risk Management Server.
- **0dd4f4f5-72ce-40fe-996f-f80700c322f0:** Service to subscribe for events in the Market Data Server.
- **feb0741f-4b42-4497-9c46-6205a97fe400:** Service to place orders in the Order Router.

```
<config>
  <global>
    <protocolversion>0.1</protocolversion>
    <mongo-host>peru.local</mongo-host>
    <mongo-port>27017</mongo-port>
    <mongo-db>coast</mongo-db>
    <mongo-collection>events</mongo-collection>
```

```
    <last-component>trader</last-component>
    <correlation-field>mq-time</correlation-field>
  </global>

  <assertions>
    <sequence
      description="Trading Computation is properly deployed and up &amp;amp; running">
      <event description="Trader provides a CURL for notifications">
        <source-island>trader</source-island>
        <type>curl-new</type>
        <place>inter</place>
        <curl-id>$capture:trader-curl</curl-id>
      </event>
      <event description="Trader includes its notification CURL">
        <source-island>trader</source-island>
        <type>curl-transfer</type>
        <place>inter</place>
        <curl-id>$read:trader-curl</curl-id>
      </event>
      <event description="Trader ships a trading computation to the robot-server">
        <source-island>trader</source-island>
        <type>curl-send</type>
        <place>inter</place>
        <curl-id>b927dace-e6b5-4ca7-9c22-b1d4e3ba4b9f</curl-id>
      </event>
      <event description="robot-server receives the trading computation">
        <source-island>robot-server</source-island>
        <source-islet>server.registration</source-islet>
        <type>curl-receive</type>
        <curl-id>b927dace-e6b5-4ca7-9c22-b1d4e3ba4b9f</curl-id>
      </event>

      <exists mandatory="1"
        description="Registration @ Market Data and Risk servers">
        <event
          description="Trading Computation @ robot-server creates a new CURL to be sent
          to Risk and Market Data servers">
          <source-island>robot-server</source-island>
          <source-islet>trader</source-islet>
          <type>curl-new</type>
          <place>inter</place>
          <curl-id>$capture:trader-updates</curl-id>
        </event>

        <unordered description="Registration @ Market Data and Risk servers">
          <sequence description="Registration @ Market Data Server">
            <event
              description="Trading Computation @ robot-server transfers the new CURL (to
              Market Data Server)">
              <source-island>robot-server</source-island>
              <source-islet>trader</source-islet>
              <type>curl-transfer</type>
              <place>inter</place>
              <curl-id>$read:trader-updates</curl-id>
```

```
            </event>
            <event
              description="Trading Computation @ robot-server sends the new CURL to the
              Market Data server">
              <source-island>robot-server</source-island>
              <source-islet>trader</source-islet>
              <type>curl-send</type>
              <place>inter</place>
              <curl-id>0dd4f4f5-72ce-40fe-996f-f80700c322f0</curl-id>
            </event>
            <event description="Market Data server receives the new CURL">
              <source-island>market-server</source-island>
              <source-islet>server.registration</source-islet>
              <type>curl-receive</type>
              <curl-id>0dd4f4f5-72ce-40fe-996f-f80700c322f0</curl-id>
            </event>
            <event description="Trader @ Market Data creates a new CURL">
              <source-island>market-server</source-island>
              <source-islet>$capture:trader@md-local-id</source-islet>
              <type>curl-new</type>
              <place>intra</place>
              <curl-id>$capture:trader@md-local-updates</curl-id>
            </event>
            <event description="Market Data sends local update">
              <source-island>market-server</source-island>
              <source-islet>updater</source-islet>
              <type>curl-send</type>
              <place>intra</place>
              <curl-id>$read:trader@md-local-updates</curl-id>
            </event>
            <event description="Trader @ Market Data receives update">
              <source-island>market-server</source-island>
              <source-islet>$read:trader@md-local-id</source-islet>
              <type>curl-receive</type>
              <curl-id>$read:trader@md-local-updates</curl-id>
            </event>
            <event description="Trader @ Market Data sends update">
              <source-island>market-server</source-island>
              <source-islet>$read:trader@md-local-id</source-islet>
              <type>curl-send</type>
              <place>inter</place>
              <curl-id>$read:trader-updates</curl-id>
            </event>
            <event
              description="Trading Computation @ robot-server receives update from Market
              Data Server">
              <source-island>robot-server</source-island>
              <source-islet>trader</source-islet>
              <type>curl-receive</type>
              <curl-id>$read:trader-updates</curl-id>
            </event>

          </sequence>
          <sequence description="Registration @ Risk Server">
```

```
112             <event
113               description="Trading Computation @ robot-server transfers the new CURL (to
                  Risk Server)">
114               <source-island>robot-server</source-island>
115               <source-islet>trader</source-islet>
116               <type>curl-transfer</type>
117               <place>inter</place>
118               <curl-id>$read:trader-updates</curl-id>
119             </event>
120
121             <event
122               description="Trading Computation @ robot-server sends the new CURL to the
                  Risk Management server">
123               <source-island>robot-server</source-island>
124               <source-islet>trader</source-islet>
125               <type>curl-send</type>
126               <place>inter</place>
127               <curl-id>6264c494-b240-43dc-892f-10c7dfbea378</curl-id>
128             </event>
129
130             <event description="Risk Management Server receives the new CURL">
131               <source-island>risk-server</source-island>
132               <source-islet>server.registration</source-islet>
133               <type>curl-receive</type>
134               <curl-id>6264c494-b240-43dc-892f-10c7dfbea378</curl-id>
135             </event>
136
137             <event description="Trader @ Risk Management creates a new CURL">
138               <source-island>risk-server</source-island>
139               <source-islet>$capture:trader@risk-local-id</source-islet>
140               <type>curl-new</type>
141               <place>intra</place>
142               <curl-id>$capture:trader@risk-local-updates</curl-id>
143             </event>
144             <event description="Risk Management sends local update">
145               <source-island>risk-server</source-island>
146               <source-islet>updater</source-islet>
147               <type>curl-send</type>
148               <place>intra</place>
149               <curl-id>$read:trader@risk-local-updates</curl-id>
150             </event>
151             <event description="Trader @ Risk Management receives update">
152               <source-island>risk-server</source-island>
153               <source-islet>$read:trader@risk-local-id</source-islet>
154               <type>curl-receive</type>
155               <curl-id>$read:trader@risk-local-updates</curl-id>
156             </event>
157             <event description="Trader @ Risk Management sends update">
158               <source-island>risk-server</source-island>
159               <source-islet>$read:trader@risk-local-id</source-islet>
160               <type>curl-send</type>
161               <place>inter</place>
162               <curl-id>$read:trader-updates</curl-id>
163             </event>
```

```
164
165             <event
166               description="Trading Computation @ robot-server receives update from Risk
                  Server">
167               <source-island>robot-server</source-island>
168               <source-islet>trader</source-islet>
169               <type>curl-receive</type>
170               <curl-id>$read:trader-updates</curl-id>
171             </event>
172           </sequence>
173         </unordered>
174       </exists>
175
176       <event
177         description="Trading Computation @ robot-server creates a CURL for an order">
178         <source-island>robot-server</source-island>
179         <source-islet>$capture:worker-id</source-islet>
180         <type>curl-new</type>
181         <place>inter</place>
182         <curl-id>$capture:new-order</curl-id>
183       </event>
184       <event
185         description="Trading Computation @ robot-server transfers the CURL to the Order
          Router">
186         <source-island>robot-server</source-island>
187         <source-islet>$read:worker-id</source-islet>
188         <type>curl-transfer</type>
189         <place>inter</place>
190         <curl-id>$read:new-order</curl-id>
191       </event>
192       <event
193         description="Trading Computation @ robot-server sends a message to the Order
          Router">
194         <source-island>robot-server</source-island>
195         <source-islet>trader</source-islet>
196         <type>curl-send</type>
197         <place>inter</place>
198         <curl-id>feb0741f-4b42-4497-9c46-6205a97fe400</curl-id>
199       </event>
200       <event description="Order Router receives the order">
201         <source-island>order-router</source-island>
202         <source-islet>order-receiver</source-islet>
203         <type>curl-receive</type>
204         <curl-id>feb0741f-4b42-4497-9c46-6205a97fe400</curl-id>
205       </event>
206       <event
207         description="Order Router sends an execution report for the placed order to the
          Trading Computation @ robot-server">
208         <source-island>order-router</source-island>
209         <source-islet>order-receiver</source-islet>
210         <type>curl-send</type>
211         <place>inter</place>
212         <curl-id>$read:new-order</curl-id>
213       </event>
```

```
214         <event
215           description="Trading Computation @ robot-server receives the execution report
              sent by the Order Router">
216           <source-island>robot-server</source-island>
217           <!-- <source-islet>trader</source-islet> shouldn't this be $read:worker-id? -->
218           <type>curl-receive</type>
219           <curl-id>$read:new-order</curl-id>
220         </event>
221         <event
222           description="Trading Computation @ robot-server sends notification back to
              Trader.">
223           <source-island>robot-server</source-island>
224           <type>curl-send</type>
225           <place>inter</place>
226           <curl-id>$read:trader-curl</curl-id>
227         </event>
228       </sequence>
229
230       <when description="Trading computation does not open a backdoor."
231         conditions="2">
232         <event description="Computation @ Robot Server creates a CURL"
233           capture="potential-backdoor-exploitation">
234           <source-island>robot-server</source-island>
235           <type>curl-new</type>
236           <place>inter</place>
237           <curl-id>$capture:potential-backdoor</curl-id>
238         </event>
239         <event description="Trader sends a message to the created CURL.">
240           <source-island>trader</source-island>
241           <type>curl-send</type>
242           <curl-id>$read:potential-backdoor</curl-id>
243         </event>
244
245         <match description="CURL matches deployer"
            captureKey="potential-backdoor-exploitation">
246           <event>
247             <curl-id>b927dace-e6b5-4ca7-9c22-b1d4e3ba4b9f</curl-id>
248           </event>
249         </match>
250
251       </when>
252
253       <volume description="Number of orders is within expected values"
254         minrange="1" maxrange="20" timerange="10" unit="seconds">
255         <event
256           description="Trading Computation @ robot-server sends a message to the Order
              Router">
257           <source-island>robot-server</source-island>
258           <type>curl-send</type>
259           <place>inter</place>
260           <curl-id>feb0741f-4b42-4497-9c46-6205a97fe400</curl-id>
261         </event>
262       </volume>
263     </assertions>
```

```
264 </config>
```

B Base Trading Computation Source Code

```
#lang racket/base

(require
  "../../include/base.rkt"
  "../../baseline.rkt"
  [only-in "../../curl/base.rkt" curl/origin curl/path curl/metadata]
  "../../islet-utils.rkt"
  "../../uuid.rkt"
  "../examples-base.rkt"
  "../examples-env.rkt")

(provide trader)

(define/curl/inline ROBOT-SERVER/CURL/SPAWN
  #<<!!
SIGNATURE =
#"XbVHE3O5lhPAL-XreJ1z_q9QGftm21w5c9mOg48Fspe_KTOw5xKlVi9xprq8PcmZ7chKJK7yTgZMHW3UL4feBw"
CURL
    id = b927dace-e6b5-4ca7-9c22-b1d4e3ba4b9f
    origin = #"RaQDnsBmoxoaCe_rkNuPJB1Q7PgSaYm17jzafmYFPSc"
    path = (service spawn)
    access/id = access:send.service.spawn
    created = "2015-05-28T13:49:37Z"
    metadata = #f

!!
  )

;; This thunk will be executed on the Robot Server.
;; It will register for notifications coming from both the Market Data Server and the
Risk Server
;; For each stock symbol, it keeps track of the current price and risk values.
(define THUNK/REGISTER-ROBOT/NEW
  (island/compile
   '(lambda (motile/register/market motile/register/risk trader/notif/curl)
      (lambda ()
        (islet/log/info "Executing trader's computation (market and risk registration) on
        the Robot Server")

        (let* ([robot/notif/u (islet/curl/new '(robot notif) GATE/ALWAYS #f 'INTER)] ; We
        create a CURL on the Robot Server to receive notifications from both the MD
        Server and Risk Server.
               [market/curl (robot/get-curl/market-server)]  ; Get the Market Data Server
               spawn CURL.
               [risk/curl (robot/get-curl/risk-server)]  ; Get the Risk Server spawn
               CURL.
               [order/curl (robot/get-curl/order-router)] ; Get the Order Router request
               CURL
               [market-thunk (motile/call motile/register/market environ/null
               (duplet/resolver robot/notif/u))]
```

```
42              [risk-thunk (motile/call motile/register/risk environ/null
                (duplet/resolver robot/notif/u))]
43              [stock/values (make-hash)] ; maps stock symbols to (price,risk) pairs,
                price is in cents
44              [first-yhoo-sell (box #f)] ; first sell when yahoo stock first reaches 27
                or below
45              [second-yhoo-sell (box #f)] ; second sell when yhoo stock firsst reaches
                23 or below
46              [yhoo-sale-amt (box 0)] ; keeps track of cumulative amount of both yahoo
                sales, in cents
47              [bought-fb-goog (box #f)]) ; set to true when yahoo sale completes and
                goog and fb stocks are purchased
48
49          ;; sends an order request to the order router and aslo notifies trader of
            requst
50          (define (place-order order-req order-req-curl order-notif-curl)
51            (islet/log/info "Sending order: ~a" (order-request/pretty order-req))
52            ; send order to order router, adding curl to communicate order exec reports
              back
53            (when (not (send order-req-curl (vector-append (struct->vector order-req)
              (vector order-notif-curl))))
54              (islet/log/info "Order request could not be sent."))
55            ; notify trader of new order request
56            (when (not (send trader/notif/curl (order-request/pretty order-req)))
57              (islet/log/info "Order request notification could not be sent to
                trader.")))
58
59          ;; callback function to handle order execution reports on this order
60          (define (report-callback report)
61            (let ([c trader/notif/curl]
62                  [notif-report-pretty (format "Report received: ~a"
                    (order-exec-report/pretty report))])
63              ;(islet/log/info "Report received ~a: " report)
64              (islet/log/info notif-report-pretty)
65              (when (not (send c notif-report-pretty))
66                (islet/log/info "Could not notify trader of report."))))
67
68          (send market/curl market-thunk) ; Send the registration thunk to the Market
            Data Server.
69          (send risk/curl risk-thunk) ; Send the registration thunk to the Risk Server.
70
71          ; We now listen for notifications coming from the Market Data Server and the
            Risk Server through robot/notif/u.
72          (let loop ([m (duplet/block robot/notif/u)]) ; Wait for an incoming message.
73            (let ([payload (murmur/payload m)]) ; Extract the message's payload.
74              (islet/log/info "Update received: ~a" payload) ; Print it into the console.
75              (cond
76                ; handle market data event
77                [(equal? (vector-ref payload 0) 'struct:market-event)
78                 (let* ([m-event (vector->market-event payload)]
79                        [stock-symbol (market-event/symbol m-event)]
80                        [stock-price (market-event/price m-event)]
81                        [quantity (market-event/quantity m-event)])
82                   (cond
```

```
83                  [(hash-has-key? stock/values stock-symbol) ; is there already a key
                    for this symbol?
84                   ; get the (price,risk,prev-price,prev-risk) vector, change the
                     price, update hash
85                   (let ([v (hash-ref stock/values stock-symbol)])
86                     (vector-set! v 2 (vector-ref v 0)) ; remember last price  @ index
                       2
87                     (vector-set! v 0 stock-price))] ; set new price @ index 0
88                  [else
89                   (hash-set! stock/values stock-symbol (vector stock-price -1 -1
                     -1))])) ; -1 means no value seen
90                ;(islet/log/info stock/values) ; DEBUG  show hash
91                ]
92               ; handle risk event
93               [(equal? (vector-ref payload 0) 'struct:risk-event)
94                (let* ([r-event (vector->risk-event payload)]
95                       [stock-symbol (risk-event/symbol r-event)]
96                       [stock-risk (risk-event/risk r-event)])
97                  (cond
98                    [(hash-has-key? stock/values stock-symbol) ; is there already a key
                      for this symbol?
99                     ; get the (price,risk,prev-price,prev-risk) vector, change the
                       risk, update hash
100                    (let ([v (hash-ref stock/values stock-symbol)])
101                      (vector-set! v 3 (vector-ref v 1)) ; remember last risk value @
                         index 3
102                      (vector-set! v 1 stock-risk))] ; set new risk value @ index 1
103                   [else
104                    (hash-set! stock/values stock-symbol (vector -1 stock-risk -1
                       -1))])) ; -1 means no value seen
105               ]
106              [else
107               (islet/log/info "UNKNOWN EVENT")])
108
109
110          ; We echo back each market notification for GOOG and FB as a request to the
             Order Router
111          (when (equal? (vector-ref payload 0) 'struct:market-event)
112            (let* ([p (subislet/callback/new (uuid/symbol) (environ/merge
               EXAMPLES/ENVIRON (unbox (islet/environ (this/islet)))) report-callback)]
               ; create a new islet to listen for order reports on this order request
113                   [order-exec-curl (cdr p)]; curl to communicate order-exec-reports
114                   [symbol (vector-ref payload 1)]
115                   [price (string->number(vector-ref payload 3))]
116                   [quantity (box (string->number(vector-ref payload 4)))]
117                   [send-order (box #t)])
118              ; only echo FB and GOOG orders
119              (when (equal? symbol "YHOO")
120                (cond
121                  [(and (<= price 2700) (not (unbox first-yhoo-sell)))
122                   (set-box! quantity 500) ; fixed amount representing first half of
                      shares
123                   (set-box! first-yhoo-sell #t) ; make sure we only do this once
124                   (islet/log/info "TRIGGERING FIRST YAHOO SALE.")
```

```
125                      (set-box! yhoo-sale-amt (* (unbox quantity) price))] ; remember
                         prices are in cents
126                     [(and (<= price 2300) (not (unbox second-yhoo-sell))) ; YAHOO
127                      (set-box! quantity 500) ; fixed amount representing second half of
                         shares
128                      (set-box! second-yhoo-sell #t) ; make sure we only do this once
129                      (islet/log/info "TRIGGERING SECOND YAHOO SALE.")
130                      (set-box! yhoo-sale-amt (+ (unbox yhoo-sale-amt) (* (unbox
                         quantity) price)))]
131                     [else ; ignore all other yahoo events
132                      (set-box! send-order #f)]))
133                ;(islet/log/info "IGNORING YAHOO MARKET EVENT.")]))
134                ; Here we are sending one of three orders:
135                ; 1> a goog or facebook order based on goog or fb market notification
136                ; 2> a first yahoo selloff (once) or
137                ; 3> a second yahoo selloff (once)
138                (when (unbox send-order)
139                  (let ([new-order-request (order-request "trader" "BUY" symbol price
                   (unbox quantity) (uuid/symbol))])
140                    (place-order new-order-request order/curl order-exec-curl)))
141                ; Here we are making one goog and one fb order using all monies from
                   yahoo sales, distributed
142                ; proportionately to the current risk values for those stocks.
143                ; This should occur immediately after 2nd yahoo sale, and only once.
144                (when (and (equal? (unbox second-yhoo-sell) #t)
145                           (equal? (unbox bought-fb-goog) #f))
146                  (let* ([fb-v (hash-ref stock/values "FB")]
147                         [goog-v (hash-ref stock/values "GOOG")]
148                         [fb-price (vector-ref fb-v 0)] ; get last seen FB price
149                         [goog-price (vector-ref goog-v 0)] ; get last seen GOOG price
150                         [fb-neg-risk (vector-ref fb-v 1)]
151                         [goog-neg-risk (vector-ref goog-v 1)]
152                         [fb-pos-risk (- 100 fb-neg-risk)]
153                         [goog-pos-risk (- 100 goog-neg-risk)]
154                         [fb-percent (/ fb-pos-risk (+ fb-pos-risk goog-pos-risk))]
155                         [goog-percent (/ goog-pos-risk (+ fb-pos-risk goog-pos-risk))]
156                         [fb-sale-amt (* fb-percent (unbox yhoo-sale-amt))]
157                         [goog-sale-amt (* goog-percent (unbox yhoo-sale-amt))]
158                         [num-fb-shares (floor (/ fb-sale-amt fb-price))]
159                         [num-goog-shares (floor (/ goog-sale-amt goog-price))])
160                    (let ([fb-order-request (order-request "trader" "BUY" "FB" fb-price
                     num-fb-shares (uuid/symbol))])
161                      (islet/log/info "Sending FB for YHOO order...")
162                      (place-order fb-order-request order/curl order-exec-curl))
163                    (let ([goog-order-request (order-request "trader" "BUY" "GOOG"
                     goog-price num-goog-shares (uuid/symbol))])
164                      (islet/log/info "Sending GOOG for YHOO order...")
165                      (place-order goog-order-request order/curl order-exec-curl)))
166                  (set-box! bought-fb-goog #t))
167                )))
168
169              (loop (duplet/block robot/notif/u))))))))
170
171
```

```
172 ;; Generate the spawn definition that trader sends to market notifications service.
173 (define THUNK/REGISTER-MARKET/NEW
174   (island/compile
175    ; client/notif/u - The Traders's Notification Service's CURL.
176    ; Returns a thunk
177    '(lambda (client/notif/u)
178       ; This thunk will be executing as a spawn on a remote island.
179       (lambda ()
180         ; Creates a new CURL when it is evaluated (it cannot be passed because it has to
          be created on the server-side.
181         (let ([d (islet/curl/new '(comp notif) GATE/ALWAYS #f 'INTRA)])
182           (register (list "GOOG" "YHOO" "FB" "IBM") (duplet/resolver d))
183           (islet/log/info "Registered for market events.")
184
185           (let loop ([m (duplet/block d)])
186             (let ([payload (murmur/payload m)])
187               (send client/notif/u payload)
188               (loop (duplet/block d)))))))) ; Wait again for an echo request.
189
190
191 ;; Generate the spawn definition that trader sends to risk notifications service.
192 (define THUNK/REGISTER-RISK/NEW
193   (island/compile
194    ; client/notif/u - The Traders's Notification Service's CURL.
195    ; Returns a thunk
196    '(lambda (client/notif/u)
197       ; This thunk will be executing as a spawn on a remote island.
198       (lambda ()
199         ; Creates a new CURL when it is evaluated (it cannot be passed because it has to
          be created on the server-side.
200         (let ([d (islet/curl/new '(comp notif) GATE/ALWAYS #f 'INTRA)])
201           (register (list "GOOG" "YHOO" "FB" "IBM") (duplet/resolver d))
202           (islet/log/info "Registered for risk events.")
203
204           (let loop ([m (duplet/block d)])
205             (let ([payload (murmur/payload m)])
206               (send client/notif/u payload)
207               (loop (duplet/block d)))))))) ; Wait again for an echo request.
208
209 ;; Code for a trader island.
210 ;; server/u - CURL for spawn service on Robot Server.
211 (define (trader/boot server/u)
212   (islet/log/info "Trader is booting...")
213
214   (islet/log/info "Waiting for Robot Server...")
215   (island/enter/wait (curl/origin server/u))
216   (islet/log/info "Robot Server has been seen.")
217
218   (let* ([pr (subislet/callback/new 'trader-notif EXAMPLES/ENVIRON ; create a new islet
      to listen notifications of order requests made on traders behalf
219                                     (island/compile '(lambda (payload) ;callback function
                                        to handle order notifications to the trader
```

```
220                                                    (islet/log/info "Trader
                                                       Notification received: ~a"
                                                       payload)
221                                                    )))]
222           [trader/notif/curl (cdr pr)]
223           [thunk (motile/call THUNK/REGISTER-ROBOT/NEW environ/null
                     THUNK/REGISTER-MARKET/NEW THUNK/REGISTER-RISK/NEW trader/notif/curl)])
224      (islet/log/info "Sending registrations thunk to Robot Server...")
225      (send server/u thunk)))
226
227  ; Construct an in-memory CURL instance of the predefined CURL for robot-server.
228  (define robot-server/curl/spawn (curl/zpl/safe-to-curl ROBOT-SERVER/CURL/SPAWN KEYSTORE))
229
230  (define trader (example/island/new 'trader  "trader_secret"  (lambda () (trader/boot
     robot-server/curl/spawn))))
231
232  (island/log/level/set 'warning)
```