Technische Universität Hamburg-Harburg
Institute for Software Systems

Run-Time Load Analysis of Multi-Threaded Applications by Inspection of Inter-Process Communication

Bachelor Thesis

submitted by
Nikolai Weh

supervised by
Prof. Dr. Sibylle Schupp

Eidesstattliche Erklärung

Ich, Nikolai Weh, erkläre hiermit an Eides Statt, dass die vorliegende Bachelorarbeit mit dem Titel *Run-Time Load Analysis of Multi-Threaded Applications by Inspection of Inter-Process Communication* selbständig und lediglich unter Benutzung der angegebenen Hilfsmittel und Literatur angefertigt wurde.

Ich erkläre weiterhin, dass die vorliegende Arbeit nicht im Rahmen eines weiteren Prüfungsverfahrens vorgelegt wurde.

Hamburg, der _____

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to thank Prof. Dr. Sibylle Schupp, head of the Institute for Software Technology Systems, and Dr. Joachim Worringen at IAT for giving me the opportunity to work on this project and their valuable assistance.

I would also like to thank IAT International Algorithmic Trading GmbH for sponsoring this research project, and all of my colleagues in the software engineering department for their support.

Finally, I would like to thank Maren, my family and my friends for their support and understanding.

Abstract

In a concurrent system, where threads are connected by queues in a pipelined scheme, the queue fill level could give information about possible bottlenecks and performance issues. This thesis presents the design, implementation, and validation of an approach that allows runtime observation of these fill levels in the context of an application focused on low latency. The approach described is therefore focused on having little performance impact itself. A basic implementation for this approach is given, along with suggestions for improvements and alternatives for specific parts of the approach. The implementation is then tested for accuracy, performance and meaningfulness of the gained data regarding performance analysis. It is shown that in a system with focus on low-latency, the fill levels of the queues are mostly zero, but that peak values can indicate performance problems.

1 Introduction

1.1 Problem Statement

Measuring application performance is an important part in an application's lifecycle. It is required when implementing the application, while planning deployment of an application, and to determine run-time behavior in production. In a concurrent software application, where threads are connected in a pipe-lined scheme using queues for communication and implicit synchronization, the queue fill level could give information about possible bottlenecks and performance issues. This thesis discusses the idea that periodically sampling the fill level of these queues could have less performance impact, can be less intrusive to the code base, and less complex to analyze than other approaches to performance analysis. The approach presented in this thesis should allow for a complete, live overview of the performance of a complex and multi-threaded application. For this purpose, I will design and implement a system designed to periodically gather information of the inter-thread communication queues' states. I will then analyze the quality of the data gained through this system, and discuss the suitability of this approach for performance analysis.

The process of determining how the software performs in multi-thread applications can be significantly more complex than the analysis of single-thread applications. However, with CPU manufacturers shifting their focus from higher clock-frequencies to greater core numbers, multi-core systems are now the standard even in the personal computer market [17]. In order to leverage the computing power of multi-core systems, application developers are forced to adopt multi-process or multi-thread techniques.

In single core applications, coarse-grained performance measuring, analysis and monitoring can essentially done by

- calculating the CPU time used by the application as well as memory access characteristics, information typically provided by the operating system,
- looking at the type and duration of I/O calls, based on information stored and managed by the VFS and network layers of the operating system, and
- using a profiler to determine which parts of the application are most frequently used and consume most execution time.

While these techniques also apply to multi-thread and multi-process applications, in these applications synchronization or locking issues may also lead to performance problems. For example, a slow[1] single-threaded application may either have too little CPU time available or is waiting for input/output completion. An application with multiple parallel strings of execution may also wait on completion of another thread of execution, thus the program may have little CPU or I/O activity, and still runs *slowly*.

Performance analysis of concurrent applications is therefore not a trivial task. How this task is approached depends on the basic architecture of the concurrency and how the threads of execution synchronize and communicate with each other. While some examples of concurrent programing usages and techniques are given

[1] In the context of this introduction, the term *slow* refers to an application that is not making full use of the system's resources.

in Section 1.4, the main focus of this work lies in the performance analysis of a system based on the Staged Event Driven Architecture (SEDA) [24], which is described in the following section.

A common approach for analyzing performance in such systems is by using the *tracing* approach. Within the system that forms the basis for this thesis, it has already been implemented for the subset of queues that form the *critical path*. This approach, however, while giving accurate results, is not a feasible option for a complete run-time performance analysis, as discussed in Section 1.3.

Therefore, another solution for a live overview of the whole system is required, possibly with lower, but sufficient accuracy and less performance impact. This could allow an operator or an algorithm to detect performance problems at runtime, and enable the tracing mechanism for the components that actually perform bad.

The approach described in this thesis focuses on analyzing the fill level of the inter-thread communication queues. A schematic overview of the base architecture for this approach is shown in Figure 1.

- Every queue instance that is related to inter-thread communication registers itself to a single registry instance.
- Additionally, the queue provides information on how it is used within the system, in order to create a graph-like representation of how the queues connect to the stages (*topology*).
- A separate thread is used to periodically request a current list of the registered queues and collects their fill level information (*sampling*).
- This information, along with the topology information, is sent to an external system for further processing.

It is expected that this approach will allow extracting of performance data while having little performance impact itself. Also, this approach relies on extending the queues rather than extending the code that uses the queues. The performance data measured can therefore automatically include even helper queues, whose performance statistics might prove valuable as well. Also, when adding new threads with the corresponding inter-thread communication, no separate performance analysis code will have to be written. This is in contrast with the tracing approach, which requires this separate code to be written whenever a queue is used. This process proved to be error-prone when working in an application with a complex layout of possible event paths. The remainder of this section is structured as follows: In 1.2, an introduction to the underlying application and its structure is given. A more in-depth introduction to the tracing approach and the problems with this is given in Section 1.3. Finally, 1.4 discusses previous performance analysis research in the area of concurrent systems.

Section 2 describes the design decisions and the implementation of the approach. In Section 3, the data gained from running the implementation in various test-scenarios is validated and analyzed, and the results regarding the quality of the gained data and its suitability for performance analysis is discussed. Finally, Section 4 concludes the thesis and suggests future work topics.

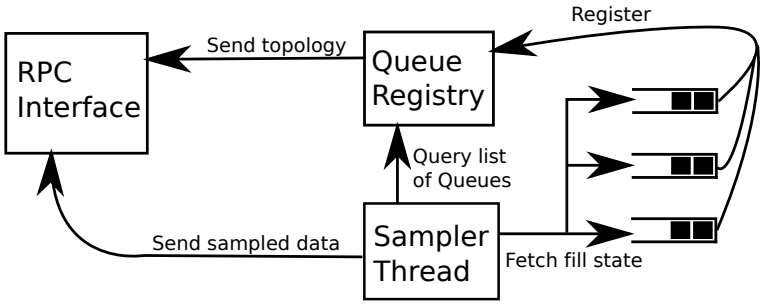

Figure 1: Queue sampling: Schematic overview

1.2 Application Layout

In this section, the **ITM** application, which forms the basis of this work, is outlined along with the requirements of a modern stock-market application and the inter-thread communication method this thesis focuses on.

The ITM application is a high-frequency trading (HFT) platform developed by IAT International Algorithmic Trading GmbH. This type of application connects to one or more stock markets, typically via multiple channels for market data and one or more connections for submission of orders and order status information. The application provides interfaces and an abstraction of the exchanges' functionality to so-called *strategies*. These react to the incoming data, to external information, and control systems used by human operators.

Application Requirements Over the past years, all major stock exchanges migrated from manual to electronic trading, so that order entry and matching is done automatically. Starting ca. 2007, this change led to the creation of high-frequency trading, which uses various *strategies* to leverage microscopic market fluctuations and differences between market places in order to make profit [19]. The second-most important factor to be able to compete, next to a decent strategy, is to minimize the latency between the market updates and the strategy, and the latency between the strategy and order entry. Minimizing latency is achieved by using server locations as close as possible to the exchanges, and state-of-the-art hardware and a software framework designed specifically for the purpose of low-latency network applications and high-frequency trading [13].

A third important aspect to consider when designing a system for high-frequency applications is that there is a special pattern of incoming data called a *microburst* [16]. A microburst can happen if a large number of participants at the exchange update their orders at the same time, and leads to a situation where the market updates appear back-to-back on the network interface. Therefore, being able to handle a (temporary) high-throughput scenario is an important design aspect of a high-frequency trading system. The microburst pattern is displayed in Figure 2, which displays the maximum amount of messages within 1 millisecond against the average message count per millisecond over one second. Shown is the average over 100 seconds. The plot has been generated from the marketdata of the Chi-X exchange on June 22, 2012.

The last important aspect is modularity, since the software should be able to

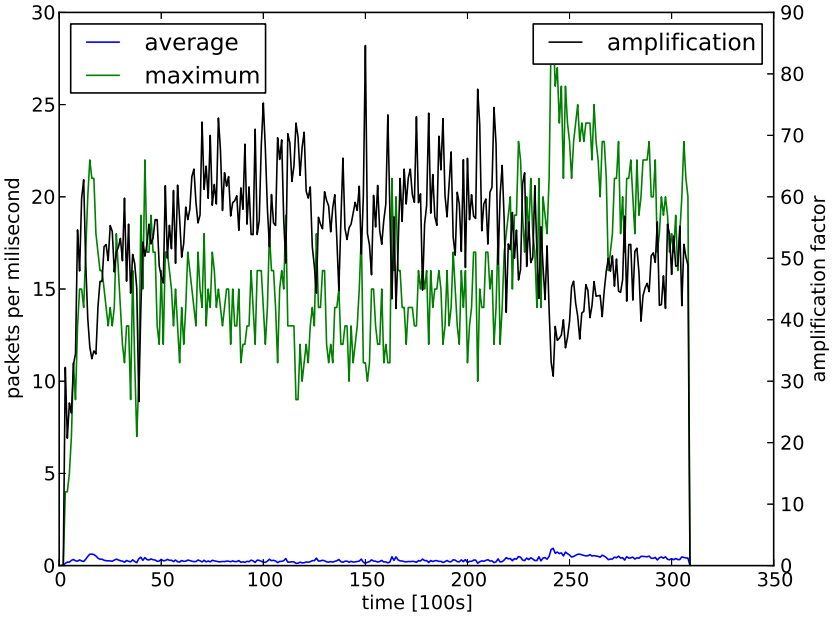

Figure 2: Microburst pattern

connect to a variety of exchanges[2] and be able to run different strategies.

Intra-Process Communication As a result of the requirements described above, the ITM has the following design characteristics:

- The system is built in a pipe-lined layout. In each logical processing step (*stage group*), work can be done in parallel by creating more than one instance (*component* or *stage*) that form a stage group.
- In each stage, an incoming event is processed, and zero or more events are generated from the resulting data. These are then passed to the instances of the next stage group.
- By leveraging the shared object functionality for the stage instance implementation, new code modules may be added at any time.
- The stages are connected using event queues. These serve both as a simple thread communication and synchronization mechanism and as a buffer in microburst situations.

[2]Although there are standards for market data messages and communication with the exchanges, nearly all exchanges choose to implement these in a non-standard way, use specific features differently, or implement their own, proprietary or "native" communication protocol. The latter is becoming increasingly popular, because it has the advantage that the exchange's software does not need to convert the data from its internal format into another, thus reducing latency.

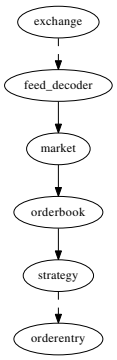

Figure 3: Critical event path

- Low latency is achieved by not relying on standard inter-thread event mechanisms[3] alone, but choosing to continuously poll the consumer interface for new data where low latency is required (*dedicated CPU core*).

The type of processing required for a stock market application dictates a standard set of stage groups, which form the *critical path* for the events with respect to the requirements listed above [13]. This critical path is shown in Figure 3. In this figure, the dashed lines show the interactions with the exchange, and the solid lines display the event flow between the stage groups. While other stage groups may exist, they are not as performance critical as those shown in the figure.

Within the ITM these stage groups are:

- *Feed Decoders*, which are used to normalize the incoming market updates into an internal format,
- the *Market*, which performs management of the orderbooks and routing of feed messages into the orderbooks,
- *Orderbooks*, which store the current prices for a specific instrument,
- *Strategies*, which apply the trading logic to the incoming events,
- *Execution Units*, which maintain orders with the exchange.

The applications design makes heavy use of multi-core systems by extensive usage of threads in (currently) 2 different processes running on a single machine. Stage instances and stage groups can be configured either as *shared*, where one thread handles multiple stage instances, or as *dedicated*, where one thread handles one single instance. Another special type of thread/instance association are worker threads. These are for example used in the *Orderbook* components. Here, multiple threads handle incoming events in parallel, partitioning the incoming and outgoing event queues between multiple threads.

[3]The standard way of waiting for a new element in the queue is using a *blocking* mechanism. If there is no new element available, the thread blocks. As soon as a new element is added to the queue, all corresponding blocking threads are woken up. While this saves CPU resources, it significantly increases latency [23].

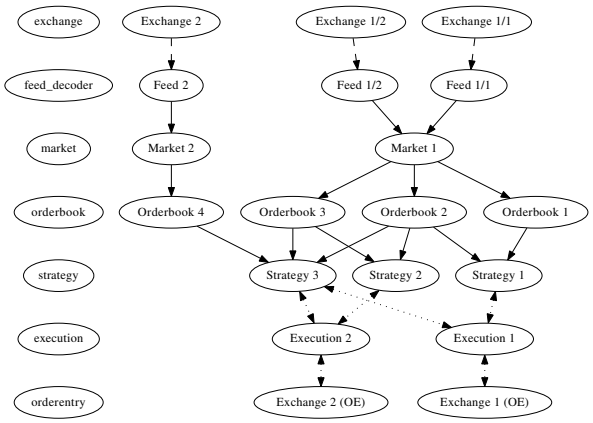

Figure 4: ITM main path layout (example)

An example configuration of the main connection path is shown in Figure 4: The left column shows the semantics of the stage group, to which the other elements belong. The graph shows a network of stages, where the edges represent the inter-thread communication queues. In this example, *Exchange 1* provides market data for three orderbooks via two channels. The three strategies subscribe to any number of orderbooks, and use one or more execution venues for order submission.

Since the execution unit is realized in a second process for security reasons, and is constructed simpler than the main process, it will not be considered in the rest of this thesis.

The usage of queues within the ITM is not limited to inter-thread communication. There are two additional queue specializations, which also share the basic *queue* interface:

- Interprocess communication queues, used for communicating work items between the main ITM process and the *Execution* process.
- Memory pools, used for assigning and releasing preallocated resources.

While these queues are not directly used for event-passing, their analysis can provide the performance analyst with additional information. The scope of this thesis does not include the analysis of such special queues. However, the design and implementation approach presented in this thesis will be discussed regarding its capability to handle future extensions, such as the inclusion of *Inter-Process Communication* (IPC) and memory pool queues.

1.3 Performance Analysis Using Tracing

This section gives a more in-depth introduction to the tracing approach to performance analysis that is already implemented within the ITM. It also describes why tracing can not be easily used for a complete coverage of the inter-thread communication.

The tracing approach delivers the most accurate results for performance analysis. Here, in each stage of the pipeline, timestamps are taken when starting

and finishing the processing of an event. The timestamps are stored in a structure, which is passed on to the next stage. The information in this structure can be extended with semantic information, such as what the incoming messages' type was, or information on how the stages handled the message. For example, a flag could be set if a conditional statement evaluated in a way that rarely-executed code was executed. Event traces can therefore give a detailed and accurate insight in the application behavior. However, some problems exist:

- Whenever an element is inserted and removed from a queue, creation and storage of a timestamp is required. Adding tracing to an existing application requires changes in the program code and data structures.
- The path of a trace may start and stop at multiple locations, making the analysis of the trace data more complex
- Using traces with some queues, such as helper queues and memory pool queues, might not be practical, as the tracing mechanism may have to be implemented separately for each use of the queue, thus introducing complexity and implementation overhead.
- Using traces introduces a considerable memory footprint and CPU usage overhead at runtime.

The problem that is most apparent when running performance analysis on a large-scale system is that the tracing mechanism has some performance impact itself. Since every time an incoming event results in multiple outgoing events, all of the previous trace data needs to be duplicated. This will result in a large amount of copy operations alongside with a large amount of redundant data that needs to be processed. In a situation with multiple hundred strategies connecting to a number of orderbooks each this might lead to notable performance degradation. *Microbursts*, as described in the previous section, amplify the problem, as longer processing time will lead to longer queues building up.

1.4 Related Work

In this section, an overview of related work on performance analysis of concurrent applications is given. Three different types and applications of concurrency are discussed. The first part focuses on work in the High Performance Computing area. The second part contains recent, popular programming models, such as the MapReduce programming model. The last part discusses work similar to, or directly related to the *SEDA* approach.

High Performance Computing Most initial research on concurrent programming and techniques was done in the context of High Performance Computing (HPC), and the area continues to be important in parallel computing research [1]. In this area, the focus lies mainly on throughput, therefore performance analysis and optimization focuses on cache and NUMA issues as well as on the optimization of the algorithms used. Most important however is that the HPC system makes best use of the computing power available, and avoids that single processes run out of work or are forced to block and wait. In order to achieve this a high parallelization factor, that is maximization of the proportion

P of the program that can be run independently from all the others, is desirable (*Amdahl's Law* [2]). Still, some form of synchronization between processes must exist, and the various approaches have generated a large amount of research activity on the performance analysis of the specific synchronization methods. In the following paragraphs, two popular approaches for thread and process synchronization in the HPC context are discussed along with performance analysis techniques.

The first technique is *message passing*, with its most popular implementation *MPI* [14]. MPI specifies an API used to exchange data between processes and distributed machines. MPI performance measurement is mostly done using event traces and collecting data from performance counters, such as message size and the time spent in MPI API calls. The MPI standard defines a profiling interface (*PMPI*, [14]), which allows profiling tools to intercept calls to MPI functions and insert instrumentation code. Examples of such tools are TAU, which focuses on automatic instrumentation tools for problem detection, and KOJAK, which provides a toolchain for instrumentation and analysis of trace data in order to perform a detailed analysis of badly performing system components [7].

The second approach is *loop-level parallelization*, where a loop statement is executed in multiple threads at the same time. The most popular framework for this is OpenMP, which requires support from the compiler used [6]. One of its main advantages is the support of an incremental development model, as it allows the introduction of parallelism to existing code with little effort [4]. There is no standard way to obtain performance statistics directly from OpenMP. Two common approaches use source-to-source transformation in order to add instrumentation code to OpenMP statements, *POMP* and *ompP* [22]. The *Thread Profiler* plugin for VTune by Intel is able to identify *OpenMP parallel regions* and provides OpenMP libraries containing instrumentation code [22].

Since the advent of multi-core multi-node clusters, hybrid programming models using both MPI and OpenMP are becoming increasingly common. Performance analysis tools and methods supporting these architectures have been developed, for example the *Scalasca performance toolset architecture* [15] and the *VampirNG* [4] infrastructure. VampirNG focuses on detailed analysis of event traces for automatic detection of problems with extensive visualization options. The Scalasca toolset uses event tracing and per-thread call-path profiling, as well as various performance counters gathered from the MPI, OpenMP, and the operating systems performance counters. Both are based on the KOJAK framework and its functionality to use compiler hooks for generation of instrumentation probes in C, C++, and Fortran.

It may be possible to perform coarse-grained analysis of asynchronous inter-process communication using the data gathered from the profiling technique in combination with event counters. However, detailed analysis of asynchronous inter-process communication with the tools and frameworks listed above is only possible by using event traces.

Recent Developments A more recently developed technology is the currently popular Google MapReduce technology, which simplifies execution of code that can be run in parallel by employing functional programming techniques. MapReduce has been designed for analysis of large datasets and is used by Google Inc., for example, for the regeneration of their web-site index. Performance

analysis therefore focuses on high throughput rather than low latency [5, 11].

A popular implementation of the MapReduce technology is the Apache Hadoop Project, on which various large-scale databases and data warehouse systems are based, such as Apache HBase and Apache Hive. Apache HBase has been a research focus of *Facebook Inc.*, which introduced optimizations regarding throughput and low latency to the HBase code base. Performance analysis can be done using event counters and timing metrics, which are exported from HBase and the *RegionServer* database manager, which is distributed with HBase [3].

SEDA Architecture A variant of the architecture used by the application that is the focus of this thesis is the *Staged Event-Driven Architecture (SEDA)* [24]. In their paper, Welsh et. al. describe a pipe-lined, event-driven layout using queues for inter-thread communication and buffering of work items. As the primary advantage, the authors cite increased *fairness* (that each incoming event is served with equal priority, and in order) and the high throughput in high-load scenarios. As an example of a problem this architecture solves in an HTTP web-server, the so-called *slashdot effect* is cited, where a website's traffic can be a increased by a factor of about 1000 for a brief period of time. In this situation, clients experience connection timeouts or partially loaded pages, because the server is unable to handle the large amount of requests in parallel. Since the SEDA approach can handle parts of the processing stages of a request sequentially or with a limited amount of parallelism, it can make more efficient use of especially disk I/O, which is hard to parallelize. This can even lead to decreased latency in comparison to a completely parallel approach, although having a large amount of requests in the input buffer and serializing requests might suggest the opposite behavior. Performance measurement of the described system is done by the clients connected to the application, and is measured in terms of response time.

Some work on on-line performance measurement of individual stages and a complete SEDA system has been done by Li et al. [20]. A system using thread pools for each stage is automatically tuned to achieve better performance by using control system techniques. For this, the number of elements inserted into the incoming event queues is used as an input for a controller, which in turn regulates the number of worker threads in each stage.

There is only little work on low-latency performance analysis and optimization. At *LMAX Ltd.*, a foreign-exchange trade platform, the so-called Disruptor-Pattern has been developed, which builds upon the SEDA architecture [23]. It is pointed out that the standard queue implementations provided by Java introduce a considerable amount of latency as well as jitter. Thompson et al. describe an alternative inter-thread communication structure that shares the *first in - first out* characteristics with queues, but is aware of CPU caches and supports multiple producers and consumers. The concurrent write access in the Disruptor pattern shares some similarities with the Phaser pattern introduced in Java 7, a deadlock-free one-way (point-to-point) synchronization mechanism (hence the name *Disruptor*). The performance analysis done by the authors is however limited to off-line analysis, using small test cases.

The approach of realizing intra-process communication by sending messages between threads through queues can be considered a low-level implementation of the *Communicating Sequential Processes* (CSP) language. This approach

forms the center of programming languages such as Newsqueak and the more recent *Go Programming Language* developed at Google, Inc. [9]. However, performance analysis utilities of these are restricted to call-graph generation and benchmarking utilities, such as the Go standard library *testing* package and the *gopprof* profiler [10].

The SEDA approach can also be considered to be an inter-thread variant of the inter-process message-queues provided by most operating systems, commonly referred to as *System V IPC* message queues. Kernel-supported distributed inter-process communication, such as the DIPC extension to the System V IPC model, has been found to yield better performance characteristics than user-space based systems such as MPI. However, performance analysis on *System V* message queues has mostly been limited to timing and throughput measurements [21, 25].

Except for the event tracing approaches, the techniques listed above focus on profiling and throughput or latency measurement. These techniques provide either off-line analysis, or coarse grained analysis of the whole system rather than of the individual components. These techniques can not be easily adapted for in-depth analysis of an asynchronous inter-process communication architecture, as required for the problem at hand.

2 Design and Implementation

In this section, the actual design and the implementation of the queue sampling method are discussed. Where applicable, alternative implementations and improvements are suggested. Four major aspects of the design are discussed. Section 2.1 contains the functionality of registering to a central instance the queues that are to be observed. In Section 2.2, topology information is added to the queues, which can be used to describe the network of queues similar to what is shown in Figure 4. In Section 2.3, the collection of the data from the queues is described. Finally, Section 2.4 describes the export of the collected data through the RPC mechanism.

This work builds upon various components that are already used and available within the ITM: There are different kinds of queue wrapper classes, which internally use either the *concurrent_queue* queue implementation made available by Intel in their Threading Building Blocks (tbb) package [18], or, in cases where less flexibility and higher performance is required, an internal ring-buffer structure developed by IAT. The RPC mechanism is implemented using the ICE framework by Zero-C [26]. This mechanism is also used by the *trace* functionality already available in the ITM.

The programming language used for creating the ITM is C++, with some performance critical parts written in C or assembly. The implementation for the queue sampling approach has been done in C++, and consists of about 480 lines of code for the queue registration and topology information classes. The implementation of the necessary RPC calls and structures consists of additional 180 lines of code.

2.1 Queue Registration

In this section, two methods for automatically registering selected queues to a central registry will be discussed. The first approach extends the base queue class by adding a second level of inheritance, thus forcing all queues to provide the interface for data sampling. The second approach provides this functionality as a *mixin*, allowing to selectively choose which derived classes should be registered and sampled. The second approach is more complex and requires programming language support for multiple inheritance, but is also more flexible than the first approach.

In order to periodically sample the queue fill levels, all relevant inter-thread communication queues must be known to a central entity. A straight-forward approach is to store pointers to the queue instances in a container structure. A sampler function must then iterate over this container, gather the fill level information, and export the data to allow analysis.

A reasonable approach for a container type would be a key-value storage such as the `std::map` container provided in the C++ STL, using an unique ID of the queue as the key, and additional information on the queue, such as the topology information described in Section 2.2, as the value. In this implementation, the memory address of the queue object is used as the queue ID. By default, elements in the `std::map` container are arranged in ascending order of the key values, which facilitates further optimizations regarding the RPC mechanism as described in Section 2.4.

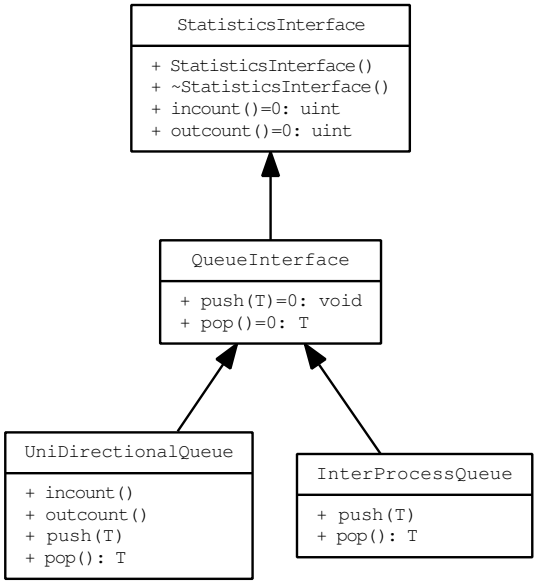

Figure 5: Queue sample interface as base class to the queue interface

Unified Interface for a Parameterized Class The design used for a central registration of all current queue instances is based on a singleton class called `QueueRegistry`. Upon construction, every queue related to inter-thread communication has to register itself to the `QueueRegistry` instance. Since multiple, different types of queues may exist and the type of data that has to be sent through the queue may differ, the queues need to inherit from one common base class providing the methods to gather the required raw data values. This base classes' constructor will also take care of registration and de-registration with the registry. A common queue interface containing the operations `push()` and `pop()` can not necessarily be extended and used for this, since these operations have a template argument or return value, thus introducing the requirement for the interface to be a template class as well. Since the queues should be stored in a single standard data container for every specialization, the basic interface must not be a template class. The approach of using a second level of inheritance, where the top-level class only provides the methods required for sampling and queue registration is shown in Figure 5. This is not a very "clean" design on one hand, since it introduces another level of inheritance but also introduces the requirement for *every* queue extending the interface to register itself with the registry. On the other hand, some special kind of queues, such as the inter-process queues used for communication with the *Execution unit*, may require separate handling when accessing statistics, or cannot provide those statistics at all. A better approach is using the statistics interface as a mixin, as shown in Figure 6. With this approach, the queue's designer may add the sample functionality where appropriate by extending an additional abstract `StatisticsInterface` class.

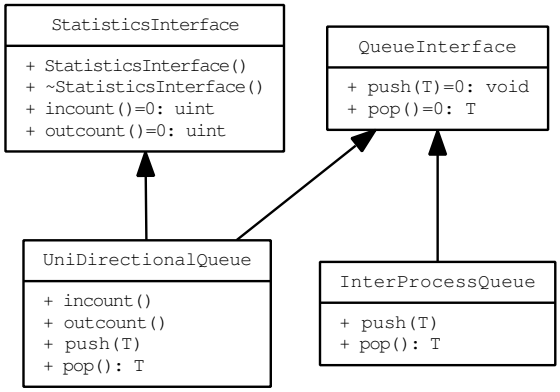

Figure 6: Queue sample interface as mixin

Handling of Special Queues Still, the problem with the registration handling of specialized queues, such as shared-memory inter-process queues, bi-directional queues, and memory pool queues, remains. These queues, while sharing a same basic interface with standard queues, may have different inner workings and can therefore require a different interface for accessing fill level or state information. A description of these queues is given below.

The first example is the memory pool queue. This class provides a number of pre-allocated objects. A thread that wishes to use an object of that type fetches one element from that queue instead of using standard memory allocation, and therefore avoids a more expensive system call along with possible non-deterministic behavior of the memory allocator. The usage information of this queue can yield performance-related data and may give insight into the application's internal processes. The results need to be derived in a different way. For example, other than with the queues connecting the stages, an empty fill level is a critical state and a full queue is not.

In the case of the shared-memory inter-process queues, it is important to consider that automatic queue registration will access the registry of the current process, or will instantiate a new registry if the current process did not yet have one.[4]

A third kind is the bi-directional queue. This kind of queue could for example extend the basic queue class by encapsulating a second "return"-queue. The queue registry needs to be aware of both encapsulated queues, and of the fact that consumer and producer switch roles when using the second queue.

To be able to handle these queues correctly, a specific mixin for each special queue can be created. This approach will allow for implementing the required functionality separately for each specialization without having to change previously created functionality. The implementation done in this thesis focuses only on queues within the critical path, which does not contain any of the special queues listed above.

[4]This is assuming a default singleton pattern, where the getInstance() call will create a new instance on demand.

Concurrency Considerations The methods of the queue registry, especially the `register` and `deregister` methods, are expected to be subject to heavy parallel usage on application startup and when changing the configuration of the system. The operations performed in both methods consist of testing whether the queue is already present within the container structure, and inserting or removing an element from the container. It is therefore expected that these methods will not use a large amount of time to finish execution. Since the creation and destruction of queues is not a performance-critical process and the methods do not invoke other methods of the same object, calls to these methods can be processed serially and locked with a non-reentrant mutex lock mechanism.

In the implementation used, when the data is read by the sampler thread, the same lock is acquired to protect the sampler thread against changes in the container structure. The iteration over the container is neither performance-critical itself, so the delay introduced by an operating `register` or `deregister` call holding the lock is acceptable, nor is it expected that the sampling itself will take time long enough to seriously interfere with application startup or when changing the configuration.

2.2 Topology Information

This section discusses methods for determining *topology* meta-information about each queue. This is the set of data containing the description on which stage instances send, and which stage instances receive events through every single queue. The advantages and disadvantages of three approaches will be discussed, where the first and the second approach directly associate queue- and thread ids, and the third approach uses a more indirect approach by adding semantic information to both queues and threads.

The topology information associates the endpoints of each queue with the type of functionality that the producers and consumers of the queue provide. It needs to show each queues place in the overall graph of edges (queues) and nodes (stages, threads). Since the queues connect threads, the thread ID may be used to create a topology map. Threads need to specify whether they are going to use a queue as a source or a drain endpoint. In order to obtain correct topology information, a thread should register if and only if it uses the queue.

The main problems when designing and implementing this functionality are:

- The thread constructing the queue can not only be either the consumer or the producer thread, but it is also possible that the queues are created by a completely different thread.
- The reverse is true as well: When constructing the threads, not every queue may be already constructed, requiring the thread to be able to detect new queues.
- New queues and threads may be added or destroyed at any time during program execution, by any other thread.
- Since latency is a constraint, updating the topology every time a `push()` and `pop()` happens is not a feasible option.
- When creating and initializing the application, constructing a new queue or when performing a standard or emergency shutdown, `push()` and `pop()`

```
while (!threadShouldTerminate()) {
  queue = selectQueueWithElements(DONT_WAIT);
  if (queue == NULL) {
    // There is currently no queue with elements waiting.
    updateTopologyInformation();
    queue = selectQueueWithElements(WAIT_UNTIL_TIMEOUT);
  }
  process(queue->pop());
}
```

Figure 7: Updating the topology information within the consumer loop

```
while (!threadShouldTerminate()) {
  if (configurationHasChanged == true) {
    updateTopologyInformation();
  }
  queue = selectQueueWithElements(WAIT_UNTIL_TIMEOUT);
  process(queue->pop());
}
```

Figure 8: Updating the topology information using a notification mechanism

operations may be performed by threads handling the current operation rather than the threads that would normally use them. For example, a memory pool queue that is filled on creation will have a number of `push()` operations done by the thread setting up the queue, and will have the same number of `pop()` operations by the thread performing the shutdown.

I have developed two basic approaches that can be used to keep an up-to-date list of queues by directly associating the queue pointers with the thread IDs.[5] The first approach implicitly updates the list of queues, after an element has been processed, and the second approach uses a notification variable, which will cause the thread to update its list of queues after a change of the queue configuration has occurred. Pseudocode-listings of how both approaches can be used in the consumer loop are given in Figure 7 and Figure 8.

Figure 7 shows a slightly optimized version of the first approach. Here, the topology information will be updated only if there is currently no further element that needs processing. The second approach is shown in Figure 8. Here, when the flag `configurationHasChanged` is set, the topology configuration is updated. In contrast to the first approach, this method does not trigger an update in every iteration. It does however rely on the queue constructor's ability to set this flag, possibly for the affected threads only, to minimize latency impact when the configuration has been changed. The latter can be hard to implement, since the

[5]Thread ID in this thesis is any unique identification for a thread. Examples of such thread IDs available in Linux could be the ID as given by the pthread system using `pthread_self()`, the operating systems' ID for the thread which is retrieved using `gettid()`, or a memory address referencing some thread specific memory structure. Which data is available depends on the thread implementation.

thread ID has to be known to the queue constructor, if the thread is constructed first.

Three main problems led to the development of a third approach. First, both approaches have an performance impact on the ITM. Limiting the extent of the impact comes with the tradeoff of not having correct topology information all the time, as the algorithm in Figure 7 may consume various elements before an update is registered. Second, mapping of more complex queue-endpoint to thread relations is not a trivial task. For example, associating a thread pool with a queue endpoint requires more complex data structures. Finally, the requirement of inserting the code in places where the queue is used by a consumer or producer, or whenever a new thread or queue is created does not meet the given requirement of the system being less intrusive to the codebase.

A third method is therefore considered, which will allow the sampler thread to request topology information when required. In this approach, semantic information about the queue is added to its constructor. This information includes for both source and drain the stage type, and an additional string parameter describing the stage instance. For example, the source stage type could be *Orderbook* and the drain type could be *Strategy*, and the string parameters could describe the feed symbol and the strategy instance ID. This approach has the advantage that not all information about the endpoints needs to be present at construction time, as the string parameter can be left blank. While this approach leads to incomplete information, it allows for a more rapid and incremental design process. It also simplifies the implementation of 1:N and N:1 relations, as described in the following paragraph.

In the case of a *thread pool* handling the queue endpoint, one provides the ID of the Pool instead of, for example, the list of thread IDs handling the endpoint. Likewise, in the case of a single thread handling a group of stage instances (*shared thread*), the queues will be grouped and identifiable by a queue group identifier. This identifier can be used as endpoint for the queues, and as *target* for the thread. This could happen in the case of a pool of worker threads that share the same queue endpoint. In a more special case that is present within the ITM, multiple pools of threads may be allocated for static subsets of the instances. For example, a single ITM instance may have a few thousand orderbooks, where each subset of 100 orderbooks is handled by a couple of threads. While this layout has the advantage of cache locality when processing the orderbooks, it makes analysis of thread load more complex.

The actual implementation used in this thesis defines a `QueueMetaInformation` class, which accepts the parameters *type* and *name* for both source and drain on construction. The queues' constructor is extended to accept an instance of this type, which is then passed to the `QueueRegistry::register` method by the `StatisticsInterface` mixin constructor. The data is then stored in the queue registry.

Since this approach allows the introduction of *blanks*, or an empty string as substitute for an correct instance identification, it allows for a fast initial implementation and iterative improvements of the accuracy of the topology information. The approach also allows extension of the topology information to queue types with special semantics, such as memory pool queues or bi-directional queues through the use of separate mixins, as described in Section 2.1.

2.3 Data Collection

In this section, the method of accessing the raw data from the queues (sampling) is described. The main problem presented here is that a lock may not be used when accessing the data, and two approaches for handling this problem are given.

Collection of the data happens periodically using a dedicated thread. This thread needs to access the list of queues and gather information:

- In-Count: The number of elements inserted into the queue,
- Out-Count: The number of elements removed from the queue and
- Timestamp: A timestamp taken directly after both previous values have been retrieved.

Since it is assumed that in a large-scale system sampling will take a measurable time, a timestamp is taken along with each data set instead of just taking one timestamp for each sample period.

There is obviously a race condition when accessing in-count and out-count without a lock. However, since the goal of this work is also to minimize latency impact, using a lock in this code is undesirable. There are two basic ways to approach this problem:

- Accept that the data can be inexact in some situations. Having the in-count sampled first and the out-count sampled last will, in some situations, lead to a negative value when calculating the element count held within the queue. The number of occurrences of these numbers may help to estimate the scale of the error introduced by the sampling thread being interrupted by the scheduler.
- Use a light-weight lock-free structure. An example is a variant of the fast-reader lock [8], where one would first read the in-count, then the out-count, and then the in-count again. If it matches the previously read in-count, a correct data snapshot has been retrieved at the time where the out-count has been read. Otherwise, the algorithm would try again until both in-counts match. This works in terms of correctness, because both values are monotonically increasing, but it introduces the possibility of a long sample duration if there is a lot of activity in the queue. A termination condition, for example after a finite number of failed tries to access the data, is therefore required.

For this implementation, the first approach has been chosen. The implementation of this functionality is also listed in Appendix A.

In order to achieve a decent quality of data while sampling, there should not be too many changes in the values between two samples. In the ITM's case, a single machine may have multiple 10 gigabit network interfaces. The size of market update messages is typically around 50 bytes, or less.[6] When a microburst occurs, these messages may appear back-to-back. Assuming a packet size of

[6]The BATS market data specification lists a few types and is available at http://www.batstrading.com/resources/membership/BATS_MC_PITCH_Specification.pdf. In version 2.0, retrieved on March 25th 2012, an "Add Order" message is between 26 and 40 bytes long, a "Modify Order" messages length can be as short as 14 bytes. No message is longer than 47 bytes.

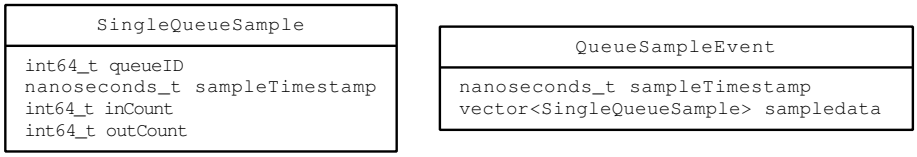

Figure 9: RPC queue sample data structure

100 byte including ethernet and protocol headers, this allows for a theoretical throughput of more than $1.25 * 10^7$ messages per second and interface. This is equivalent to one element every 100 nanoseconds. A high sample frequency will necessarily generate high load on the CPU, but will also lead to lock contention on the counter variables on cache level in extreme cases, because the cache coherency protocol will implicitly perform some sort of synchronization [12]. An evaluation of different sample frequencies is provided in Section 3.

2.4 Data Export

This section describes the final implementation stage, which consists of the export of both the topology information and the sampled data. After describing the basic implementation, which can be considered to be straight-forward, suggestions for improving the performance and the data volume are given.

The data has to be exported out of the application for performing analysis. For this, the RPC mechanism already present in the ITM is used. Along with the actual sample data, the system also needs to push to the client updates concerning the topology. In order to do this, the sampler thread needs to be aware that such a change occurred, and needs to format an update to a client. While detecting fresh queues only requires a flag in the queue (`detectedBySamplerThread`), detecting removed queues requires the sampler thread to keep a list of queues from the last sample period as well. This list would then have to be compared to the *current* list of queues in the system.

Implementation of the required functionality in the RPC mechanism is straight-forward. As an example, the RPC structure for the QueueSampleEvent is given in Figure 9. A `QueueSampleEvent` contains the timestamp when the event was generated and multiple `SingleQueueSample` structures. Each `SingleQueueSample` contains the values of the in- and out counters, read directly before the timestamp was taken, of the queue with the corresponding `QueueID`. A very basic implementation of a `TopologyUpdateEvent` would include all current `QueueIDs`, along with the endpoint types and semantic names.

Optimizations In some situations it may be necessary to reduce the data volume or the number of RPC calls to a smaller amount. A few ideas are presented here.

- **Omitting known data**: If the queue registry uses a container that is sorted by the `QueueID` and if it is certain that the topology can not change between an `TopologyUpdateEvent` and a `QueueSampledEvent`, an obvious optimization is that one can omit the `QueueID` from the data structure sent to the client. This will allow cutting the payload's size by

1/4 in the implementation used here. The client, having received a full list of `QueueID`s with the most recent `TopologyUpdateEvent` is then able to assign the correct `QueueID`s to the `SingleQueueSample` structure by sorting the list of `QueueID`s and assigning each `SingleQueueSample` one `QueueID`, in order.

- **Event coalescing**: It is possible to decouple the queue sampling from the event generation, by using a second dedicated thread for the event generation. The sampling thread would then store the results from each sample period in a data structure, and the event-generating thread would periodically collect the complete data set and send it to a client. By using a longer period than the sample period for the event-generating thread, the number of RPC calls made will be reduced.

- **Preprocessing and filtering**: Building upon the concept of the *event coalescing* above, having the data from previous sample periods available would allow preprocessing and filtering. As an example, queue samples that do not show changes when compared to the previous sample period could be omitted from the data sent via the RPC.

3 Validation

In this section, a complete application instance is tested against recorded real-world market data, using the playback speed of the recorded data, the sample frequency, and the strategy configuration as parameters. The goal is to determine how the data derived from queue sampling compares to the trace data, whether it can be used for live-performance analysis, and how the two approaches compare on a performance level. In more detail, we are looking for answers to the following:

- What is the informative value of the data generated by the queue sample data? Is it possible to observe partially filled or completely filled queue states even though the processing time of the stages is generally very short?
- Is it possible to detect performance bottlenecks and back-pressure by looking at the queue sample data? Can the queue sample data be used as a live performance overview of the whole system?
- How does the tracing mechanism react to a high number of incoming events and to a large number of concurrent traces? What is the performance of the queue sampling code when a large number of queues need to be processed?

The remainder of this section is structured as follows. First, a description of the test environment and its limitations is given, along with a description of the test procedure and parameters. Section 3.2 discusses various techniques for validating the correctness of the received data, and discusses detected discrepancies. In Section 3.3 it is discussed whether the sampling method introduces a performance penalty on the ITM at various sample frequencies. In Section 3.4 the performance of the sampling code and anomalies in the time required for sampling are analyzed. Finally, in Section 3.5, the data is tested to determine whether performance issues can be detected using the collected data, using different playback rates and strategy configurations. The validation is concluded by discussing the results of this section.

3.1 Test Environment

The following will give a description of the environment used for testing, as well as the restrictions and the differences compared to the production environment.

The tests have been performed on two *Supermicro* servers, using Intel Nehalem processors with 16 cores at 3.2 GHz clock speed and 12GB RAM each. Both are connected using two direct fiber optics connections with 10 Gigabit of throughput each. The first (*primary*) server is used to run an ITM instance. The second (*supporting*) server provides the market data as input for the primary server. Both servers run the Red Hat derivative Scientific Linux 6.2. The RPC clients are run on a third machine connected via a 1 Gigabit ethernet interface. Two different market data streams can be run through the two network interfaces, leading to a more realistic activity in the ITM.

In these tests, the market data retrieved from the exchanges BATS and Chi-X will be used as input for the so-called SmartQuota strategy. This strategy

compares prices on two exchanges, looking for minor differences and utilizes them.[7]

The choice of Chi-X and BATS was made because the feed layout is comparatively simple, and because the same instruments[8] are traded with the same symbol on both exchanges. This significantly simplifies automated testing. The market data used throughout this section was recorded on June 22, 2012 starting at 3:00 A.M. and ending at midnight. The recording has captured the whole *trading hours* phase, which ranges from 8:00 to 16:30 London time, along with the *setup* data that is sent beforehand.

An important restriction of the test environment is that the packets were streamed at a fixed rate rather than with the rate dictated by the market environment. This restriction implies that it was not possible to simulate the microburst conditions in the way they would appear in reality. Since these bursts often concentrate on a very specific subset of market data or even a single instrument, it was expected that short high-load conditions could be observed nevertheless. These would however not appear at feed level — since the incoming event speed was fixed here — but rather at later stages. A beneficial aspect of the fixed replay speed in respect to the test procedure is that tests generally run faster. For example, the data recorded from the Chi-X feed contained 9.8 million packets. This results in a test time of about 15 minutes, whereas replaying at the real speed would have resulted in a test time of 24 hours.

To to be able to partially simulate microburst conditions at feed level, a modification of the test environment is done in Section 3.5: The tool playing back the feeds is modified, so that instead of delaying the execution after every packet sent, it will only introduce the delay every Nth packet, so that N packets are sent *back-to-back* through the interfaces. This allows the simulation of bursts at the feed levels. However, these bursts are missing the semantic context between the amount of packets sent and the instruments responsible.

A second limitation became visible after initial tests had been run. It was observed that high replay-speeds resulted in dropped packets at the feed decoders. Since the feed decoders receive their packets directly from the operating system's network input buffer[9], observation of the incoming queue of the feed decoder can not be done using our approach. In a production environment, the system would request the dropped packets from a *recovery* server provided by the exchange. The test framework used provides simple replay of the recorded data, and does not provide any *recovery* functionality.

The parameters that were modified in this test are:

- **Queue Sample Period** [milliseconds]: the time the sample thread waits in between two sample runs.
- **Replay Delay** [microseconds]: configuration parameter of the packet replay program. When replaying the recorded market data streams, this is

[7] The basic idea is that if a security is sold cheaper on exchange A than it is bought on exchange B, the algorithm may buy on A and sell on B, where the difference in price times the quantity is the profit the algorithm achieves. This strategy is seldomly used in a "pure" form, since the competition here is extremely high [13].

[8] An instrument is any stock symbol, future, contract etc.

[9] On Linux, the size of the UDP input buffer depends on the available memory and the *sysctl* configuration option `net.ipv4.udp_mem`, the buffer status can be observed in the pseudo-file /proc/net/udp. More information can be found at `http://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt` (retrieved July 3, 2012).

the interval between each two consecutive packets.

- **Burst Length**: The amount of packets sent in each step by the replay utility. The default value is 1, this parameter is only changed in Section 3.5.

- **Strategy Configuration**: The default configuration uses 10 strategies that connect to one orderbook pair[10] each. In Section 3.5 another configuration is used, where 10 strategies connect to one single orderbook pair.

For each feed instance, a separate feed, market and OrderbookWorker thread is created. The strategies are configured to use the *polling queue-wait mode*, resulting in the strategy continuously polling for new data on the incoming event queues, therefore creating 100% load on one CPU core per strategy instance.

3.2 Validation of Data

In principle, it is possible to use the data generated by the tracing mechanism to verify the correctness and usefulness of the sampled data. The tracing mechanism inserts a timestamp T_{push} into the trace data vector right before an element is inserted into a queue. Right after the element has been removed, a current timestamp T_{pop} is inserted into the trace vector. Therefore, for a given timestamp T_{sample} at which the corresponding queue has been sampled, the count of traces where

$$T_{push} \leq T_{sample} \leq T_{pop}$$

holds should be equal to the measured fill level at that timestamp.

However, although this approach could work in some applications, a few assumptions have been made that may prove to be invalid:

1. It is not possible to create a timestamp at the *exact* moment of insertion into the queue. In fact, when working with nanoseconds resolution, the time spent in the required *syscall* for fetching the current time will be considerable. There is also always a timespan between the insertion of the timestamp into the vector, and the actual insertion of the element into the queue. This value can be considerably large, if additional code lines are inserted between these two actions.

2. A number of events can be filtered out at earlier stages, before the trace generation can happen. In the orderbook component, most incoming events are typically filtered out, and only those events subscribed to by the strategy get passed to the next stages.

3. When a consumer merges two or more streams, it may not be possible to determine where the origin was. This is the case with the *feed* traces within the ITM: Traces are generated when a connected orderbook sends out an event. Which orderbook sent the event, however, is not shown.

[10] For each instrument, one orderbook describes the activity of the instrument on each market. The two corresponding orderbooks on both observed markets are referred to as *orderbook pairs* in this thesis.

It is expected that the first problem will lead to some small differences in the fill level calculated through the sampling method, and the fill level calculated using data from the event traces.

Handling the second problem in this list requires that the validation is only performed at stages where one can be certain that no elements are filtered out before a trace is generated. In the ITM, this leaves us with the queues connecting *Orderbook* stages with *Strategy* stages (Orderbook→Strategy queues). A problem with these queues is that in the tests done, no non-zero fill level was visible at all. This effect will be further discussed in Sections 3.5 and 3.6.

The third problem has to be handled differently for each validation test case where this problem is relevant, and will therefore discussed separately in each test case.

The validation tests have been done using the following test setup: 10 strategies connect to 10 different orderbook pairs (*1:1 setup*). The value of 10 was chosen as a tradeoff between a large number of strategies running in parallel, and the necessity of having a few cores available for the remaining components and other operating system tasks. The replay delay has been set to 50us/packet for both interfaces, and the sample period has been set to 10ms. The instrument with the most trading activity, and therefore the one generating the most traces and the highest activity in both the orderbook and the corresponding strategy, has been chosen for the validation. Of the instruments observed, this was Adidas AG (*ADSd* on both Chi-X and BATS), which accounted for 83% (9634 of 11613) of the generated traces.

Validation of the data has been done in three steps:

Step 1: Sum of Element Counts After the measurement session, the count of traces that were received should be equal to the sum of the event counts of all incoming queues, plus the count of *control events*.[11] This is also true for every subset of the run time, in which no trace was currently active.

More specifically, the sum of all traces ending before the last queue state has been recorded should be equal to the count of consumed elements. This analysis can be done by hand when only a few queues are to be analysed.

In our measurement, the sum of eligible traces turned out to be 9634, and the Out-Counts of the most recent sample event from both orderbooks of the *ADSd*-pair are

$$C_{out}^{Chi-X} + C_{out}^{BATS} = 4405 + 5234 = 9639,$$

which leaves $9639 - 9634 = 5$ control elements, which is well within range of what was expected. For the following validation stages, these 5 elements will be implicitly subtracted from the sample data counts.

Step 2: Visual Inspection It is possible to visually inspect the increments and differences between the queue state counters and the number of traces by plotting those values against a time axis. In order to compare the cumulative count of traces with the two input streams, the sampled counts from the two input streams has to be combined. Since there is a timing difference between

[11] Control events are used within the ITM to signal the occurrence of specific events, such as packet loss or the start of the trading phase. Each stage will typically receive a small number of such events in the startup process.

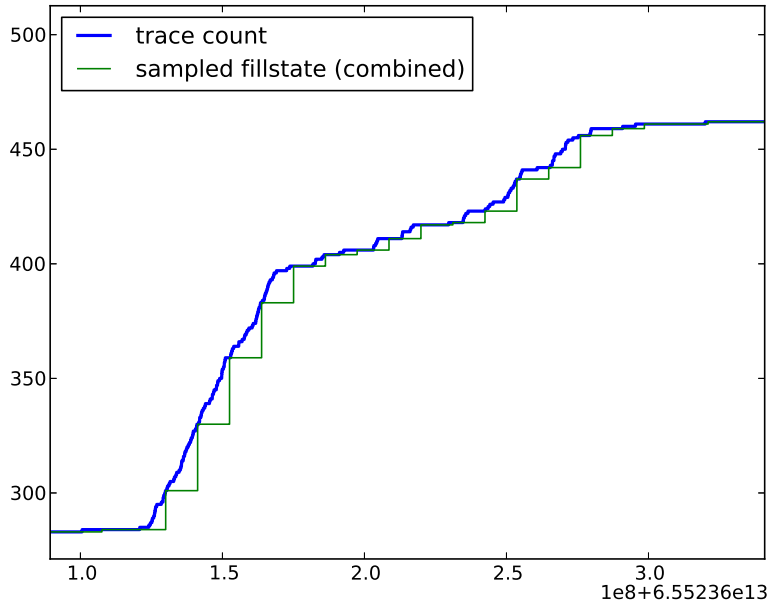

Figure 10: Trace count and sampled queue fill level over time (excerpt)

the samples of both streams, albeit little, the result of combining the counts from the same sample period might not give an exact result. It might however be sufficient for a simple evaluation. Figure 10 is an excerpt of such a plot, combining the sampled values of the two input queues while using the timestamp of the earlier sample for the x-axis. It can easily be seen that the queue sample count steps up to the trace in every sample period.

A more detailed analysis of the above can be done by including the individual timestamps from the SingleQueueSample structure in the analysis.

Step 3: Validation of Correctness The most interesting question is whether the sampled queue fill level actually reflects the count of elements at the time of sampling in the queue that is observed.

In order to perform such an analysis, a program has been created that performs the following processing on the data:

1. From the trace data, load the pair of data corresponding to the times of insertion and removal from a queue.

2. Load the sampled data from this queue and for each sample, store the timestamp T_{sample} along with the calculated fill level C.

3. Iterate over the sampled data:

   (a) Find the number of N_{match} trace pairs where the insertion timestamp is smaller and the removal timestamp is larger than T_{sample}.

(b) If N_{match} is not equal to C, output the expected and real value along with a timestamp.

The algorithm outputs the cases, where a difference between the sampled fill level and the fill level calculated using the trace data occurred. In the general case, since this program considers only one queue but the traces of two queues combined, one would expect a large number of false positives. This would happen if the trace was generated by an event that passed a queue that was not inspected. While this problem does exist, the effects are diminished here due to the already mentioned fact that on both queues a fill level > 0 could not be observed. A check whether a non-zero fill level, as calculated from the trace data, that was not detected by the sampling method in one queue, was detected by the sample date from the other queue, can therefore be omitted.

In the test scenario, the program found 17 and 13 discrepancies when analyzing the data from the Chi-X orderbook queue and the BATS orderbook queue, respectively. Manual analysis of the corresponding trace data suggested that in fact a value should have been recorded. In fact, the sampled timestamp seemed to generally be near the mean value of the insertion and removal timestamp. Since no other indication of an error in the code was given, I suspect that a scheduling problem combined with the timing discrepancy (*problem 1* described in Section 3.2) could have been responsible for the discrepancies. This theory is supported by the fact that the duration $T_{pop} - T_{push}$ was generally larger for the traces corresponding to the discrepancies. The latter was 74554 nanoseconds on average whereas the average duration over all the traces was 15572 nanoseconds.

3.3 Performance Impact

In order to measure performance impact of the queue sampling, and of using different sample frequencies, a performance test has been done. It is expected that increasing the sample frequency to a point, where the sampling thread fully utilizes one CPU core, will have a small yet measurable performance impact on the ITM.

As in the previous test, this test uses 10 strategies connected to one orderbook pair each, as a tradeoff between a large number of strategies and providing enough computing power to the remaining components. For measuring the system's performance, the data generated with the tracing mechanism has been used. Since CPU performance statistics could also be relevant, the system was run with the "perf" tool[12], which can extract various statistics from the kernel and the CPU, such as cache and TLB misses, as well as page faults and the number of instructions the program used. Since this tool causes some overhead as well, the packet delay was increased to 70μs, as lower values caused dropped packets in the feed handler. The lowest sample period possible in this test case is 5ms. Lower values caused the RPC-System to overload.[13] Other sample frequencies

[12]The perf tool has been created by Red Hat Inc. It is distributed with current Red Hat Enterprise Linux editions. More information can be obtained at `https://perf.wiki.kernel.org/index.php/Main_Page` (retrieved July 11, 2012)

[13]RPC requests are currently performed synchronously. This means that while an RPC request is processed by the client, other requests are queued. Also, when a trace is sent (with considerable payload), the queue elements will have to wait as well. Since in the current implementation heartbeat requests are queued as well, having a queue that takes more than the heartbeat timeout to process will cause the client to drop because of the missing heartbeat.

	sample freq.	mean	stddev	q1	median	q99
feed→market	5ms	62.1	153.8	17.3	55.5	172.1
	100ms	61.5	167.3	17.4	55.1	152.1
	1000ms	60.3	212.7	17.6	54.1	131.5
market→OB	5ms	56.0	55.4	13.0	54.0	128.2
	100ms	55.5	29.1	12.9	54.8	118.4
	1000ms	54.8	18.6	13.1	53.8	114.8
OB→strategy	5ms	16.8	7.0	8.0	15.5	49.7
	100ms	15.1	6.0	7.3	14.4	44.1
	1000ms	15.7	6.1	7.9	14.7	46.8

Table 1: Measured latency impacts of the queues connecting feed, market, orderbook and strategy when changing sample frequencies [μs]

used are 1000ms, which is expected to cause no measurable performance impact, and 100ms, which can be considered a tradeoff between the two other values.

For each sample frequency configuration, about 17200 traces were collected by running each test case twice.

The results are shown in Table 1. This table shows the latency introduced by the queues, as measured by the trace mechanism, for three different sample period lengths (5ms, 100ms and 1000ms) in the three critical path queues. The latencies are given as the mean value, the standard deviation (*stddev*), the 1% and 99% quantiles (*q1* and *q99*), and the median value.

As stated in Section 3.1, the Strategy components use a *polling queue-wait mode*, which leads to the significantly lower duration in the OB→strategy queues. Due to the high standard deviation shown especially in the *feed to market* queue, the quality of the results regarding the differences between various sample frequencies is questionable. It is however visible, that the mean and q99 values become significantly larger when lowering the sample period to 5ms. This is the expected behavior, since increasing the load on the memory bus will necessarily decrease the performance of other components.

No conclusive results can be derived from the q1 and median values. The standard deviation seems to become larger in the Feed→Market queues and Orderbook→Strategy queues at 5ms sample period. On the other hand, within the Feed→Market queue, a low sample period length seems have a positive effect on the q1 and stddev values. This behavior could originate from a caching effect, since having a high access rate on specific values could cause them to being kept in the CPU cache, even if they are accessed on a different CPU. Whether the observed effect can be attributed to this caching effect may be an implementation detail of the specific CPU type [12]. As shown in Table 2, using a low sample frequency seems to slightly decrease cache misses in the *Lowest Level Cache* (LLC). The accuracy of these results can however be questioned as well, since the exact behavior of the CPU caches depends on a large number of factors, and is hard to observe [12].

As the test results show a high standard variation and inconsistencies, they are not suitable for derivation of a meaningful conclusion regarding the performance impact of the sample code. As the RPC mechanism (and the sample code itself, as shown in the following section) obstruct testing at higher frequencies, it can be concluded that constructing meaningful test results is, with the given test

	LLC-load	LLC-load-misses	percentage
5ms	8.2×10^8	5.3×10^8	64.2%
100ms	7.9×10^8	5.4×10^8	68.5%
1000ms	8.1×10^8	5.5×10^8	67.4%

Table 2: LLC Load Misses at different sample frequencies

environment and implementation, not possible.

3.4 Performance of the Sampling Code

In this section, the time required for a single sample period is discussed, as the sample duration in the previous tests appeared to be unusually high. Three different theories that could be responsible are given and analyzed.

While analyzing various sample frequencies, it turned out that the duration of the sampling itself took more than 1ms for 165 queues, which can be considered to be very slow: In his paper on memory and caching [12], Drepper shows that in extreme cases, on very large datasets, access time of elements using random access can take up to 1500 CPU cycles. At 3.2 GHz CPU clock speed, this should allow for $2.13 * 10^6$ values to be accessed every second. Even in the worst-case scenario, accessing $165 * 2 = 330$ values should therefore be finished within $330/(2.13 * 10^6) = 1.5 * 10^{-4}$ seconds, or 0.15 milliseconds.

The following shows possible reasons:

1. Each single sample requires a timestamp, which will result in a *syscall*. Internal measurements done by IAT however show that the time for a call to `clock_gettime(CLOCK_REALTIME)` took 29ns on average, therefore these calls should take no longer than 5μs for all queues combined.

2. The values can not be kept in the CPU's cache due to the requirement of other threads to read and write them as well.

3. Prefetcher logic can not be used for queues that are not located in a linear layout in memory. It might even introduce performance penalties if small sets of queues are located in a linear layout, followed (in the order of sampling) by queues that are laid out differently, because the prefetcher might load memory into the cache that is not going to be used.

It was assumed that the last issue concerning the caching in a multi-core environment would account for the largest amount of time consumed. This assumption implies that queues that have low activity should take less time to be sampled, as the In-/Out-Count could stay within the sampler threads' cache. This might not apply to the strategy queues (poll-loop), depending on how the cache coherency protocol is implemented.

A test was done to verify this assumption, using 10 strategies connecting to 10 orderbook pairs in a 1:1 relation, a sample delay of 50ms, and a replay delay of 50μs. The results are shown in Figure 11. The graph shows for each queue the average sample duration (blue line) and the number of elements that were sent through the queue during the runtime (grey line, logarithmic). The first observation is that, compared to the average sample duration, some queues consistently take longer to be sampled, and some queues consistently take less

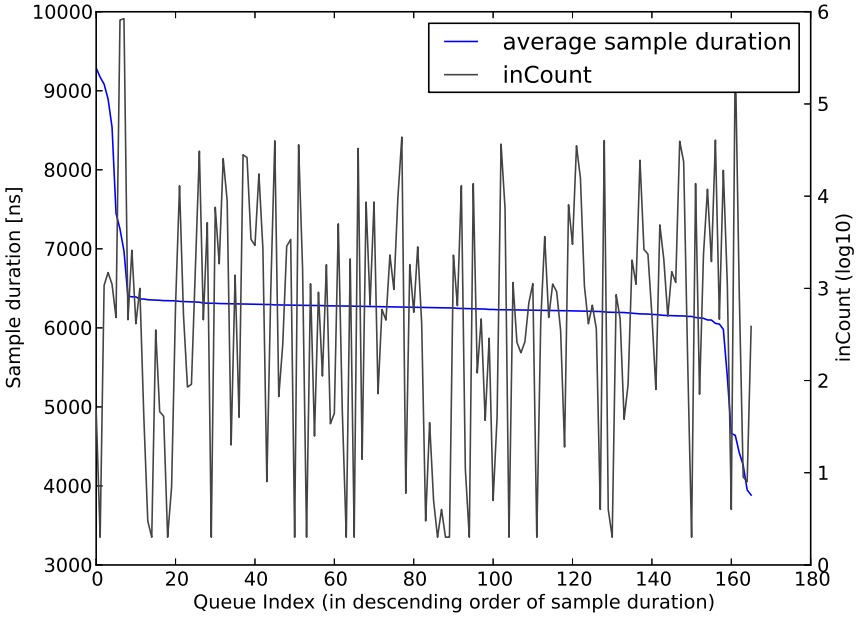

Figure 11: Queue sample duration for individual queues

time to be sampled. The next observation is that, with one exception, there does not seem to be any correlation between the number of events passing through the queue and the time required for sampling them. The exception to this is shown by the peak on the left side, which corresponds to the two *feed to market* queues. Sampling the Chi-X queue took an average of 7250ns, sampling the BATS queue 6974ns.

In order to determine whether CPU prefetcher logic could be responsible, a second graph was created. It is assumed that queues that are in a linear layout in memory, are sampled faster than queues lying at more random positions in memory. Figure 12 shows the sample duration of each queue along with the absolute difference of the memory location to the previously sampled queue. Queues are sorted by increasing memory location. The graph shows 7 peaks in sample duration over 6500, 4 of which are in a block containing about 13% of the sampled queues (Queues 145-165). In this block, the memory locations of the individual queues are non-contiguous, whereas in the remaining queues, only 2 deviations of the standard memory location difference can be observed.

Another observation that can be made in Figure 12 is that the queues with a peak maximum average queue sample duration are, in sampling order, always directly preceded by a queue with a minimum peak average queue sample duration. That this is in fact not caused by some very large deviations from the average was confirmed by calculating various quantiles, which showed similar results. Plots of Figure 11 containing various quantiles can be found in Appendix D.1.

It can be concluded that the theories presented at the beginning of this

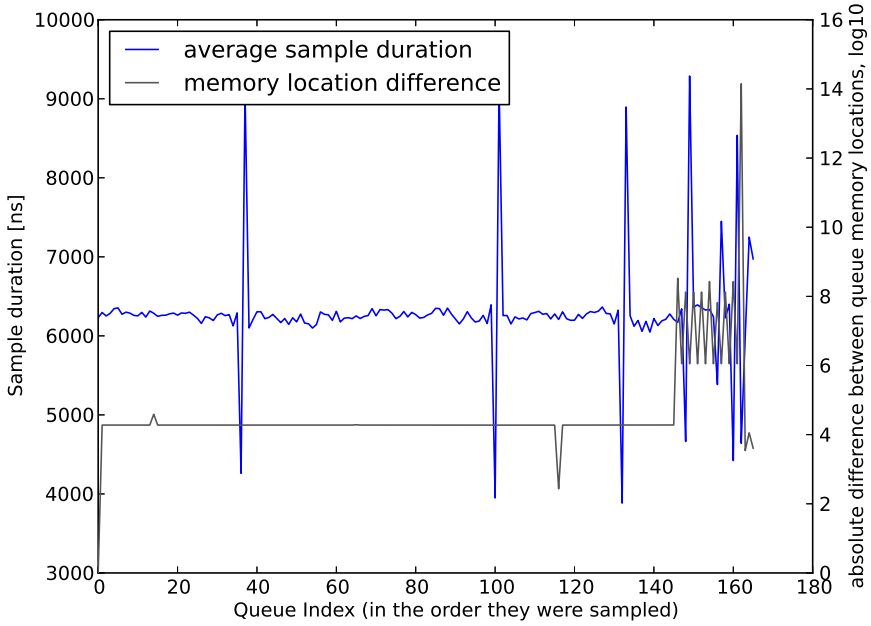

Figure 12: Queue sample duration for individual queues, in sample order

section could not be verified, and that the reason for the large sample duration remains unknown.

3.5 Visibility of Performance Data

This section focuses on the question whether it is possible to use the data as generated by the queue sampling mechanism as an indicator for performance issues. The sample interval in this test is 10ms. This value was chosen to allow for higher replay speeds compared to the tests in Section 3.3. It is first described why the queue used for communication with the RPC mechanism is excluded from the measurements. Afterwards, for each of the three major test configurations, a description of the test and an analysis of the test results is given.

RPC Queue Removal An initial inspection of the data revealed, that a distinction between the RPC queue and all other queues had to be made: RPC events in the ITM are pushed into an event queue, which is processed by a *publish-subscribe* handler every 50 milliseconds. Since every 10 milliseconds a new event is created by the sampler thread, the queue fills up with every sample period, until the event count reaches 4. In the following sample period, the queue is empty again and the pattern restarts from the beginning.

In the analysis following in the remainder of this section, the data gained from the RPC queue was removed because of the following reasons:

1. The pattern showed only slight variations, independent of what the system

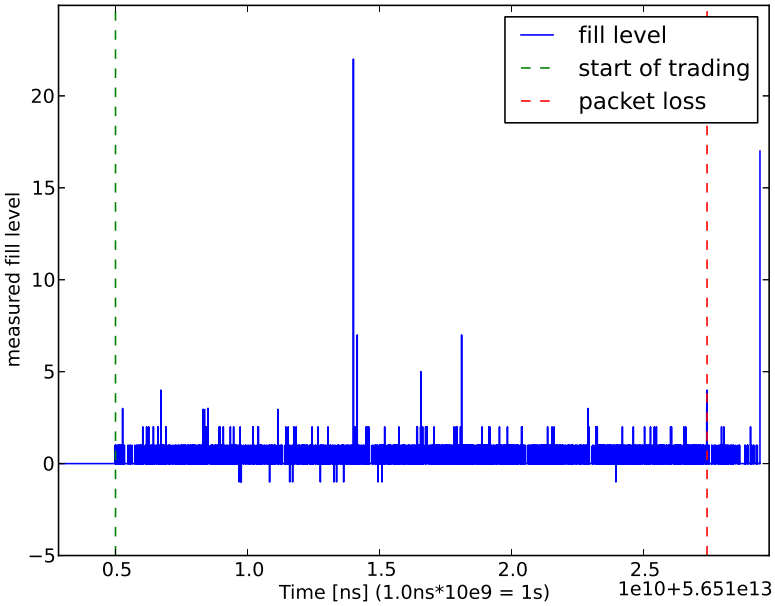

Figure 13: Queue fill levels at 40μs replay delay

load was. The variations found could either be attributed to scheduling behavior (for example, when the pattern switched from 0-1-2-3 to 1-2-3-4), or to the fact that an event has happened where clients should be notified, such as a log message or a trade event.

2. The queue does not have a size limit (`UnlimitedQueue`), so that it is not possible to fill the queue or to generate a significant amount of backpressure.

3. Since all stages share a single RPC event queue, the origin of an event can not be determined with the queue sampling method, making the data gathered unsuitable for performance analysis.

4. Events in the RPC queue tended to appear in large bursts. In combination with the fact that they are only periodically removed out of the queue, sampled values tended to become very large in some situations (especially during startup and shutdown, and at the start of trading) and overshadowed measurements from the queues in the critical path.

1:1 Orderbook-Strategy Configuration In the first test, the configuration used in the previous tests is analyzed. The first task was to determine the highest replay-speed that does not cause dropped packets in the feed decoders. This was done by manually increasing the replay delay in 10μs-increments, until a test run could be completed. The highest replay speed where the test did not fail was determined to be at a replay delay of 50μs. In order to determine if an upcoming packet loss situation was indicated by the queue sample data, the

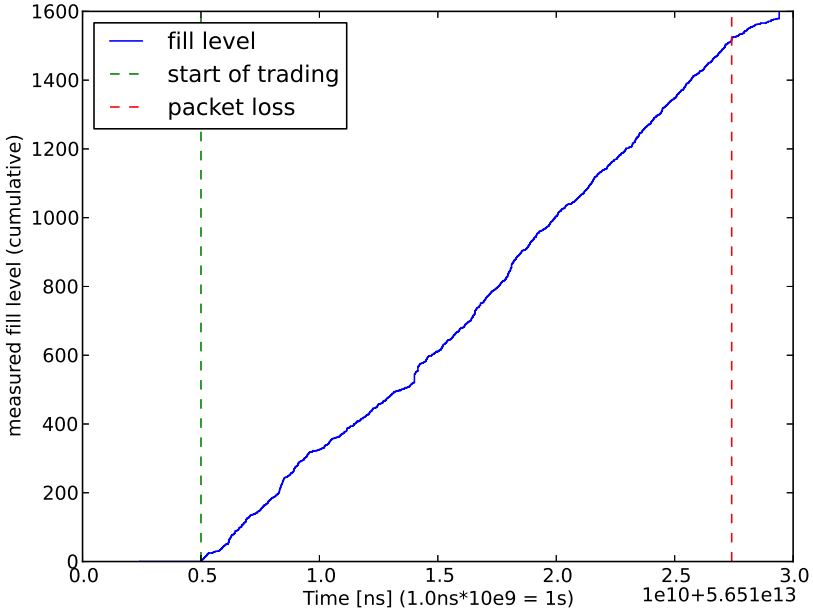

Figure 14: Queue fill levels at 40μs replay delay, cumulative sum

first analysis was done with the data gained from a test run where packet loss occurred.

Figure 13 shows the fill level plotted against the time of sample for each individual queue, using a replay delay of 40μs. The red dashed line indicates where the packet loss was detected. The green line approximates when the active trading phase started.[14]

There are 3 major observations that can be made in this graph:

- A number of small peaks occur at seemingly random positions in time. Investigation showed that the Feed→Market queues, from both feeds, are causing these peaks. The last peak can be attributed to queues filling up caused by the system shutting down.
- Two small peaks can be seen at the time of the packet loss. These are caused by the same Feed→Market queues. The fill levels are 4 and 5 for the Chi-X→Market and BATS→Market queues, respectively.
- Negative values are visible. This is an expected behavior for a highly active queue, as described in Section 2.3.

In order to determine if there are differences in the density of non-zero fill levels over time, a cumulative plot of the sum of fill levels has been created and is shown in Figure 14. This plot shows a mostly linear increase and short, larger

[14] As this information can not directly be received from the queue sample data, it is assumed that active trading has started when events are sent to an orderbook. To account for the fact that a small amount of control events is sent at startup, a threshold of 10 has been used.

Queue	count($C \geq 1$)	$\sum C$	inCount, total
Feed to Market			
Feed:BATS to Market	618	689	136393
Feed:Chi-X to Market	354	701	92803
Market to Orderbook			
to OB:Chi-X:ADSd	2	5	645
to OB:BATS:ADSd	8	8	1696
to OB:WorkerThread/1	62	40	35919
to OB:WorkerThread/0	88	89	59704

Table 3: Summary of non-zero fill levels at 40μs replay delay

Legend: count($C \geq 1$) Number of samples with non-zero fill level
$\sum C$ Sum of (non-zero) fill levels

increases at the points where Figure 13 shows peaks. No major increase of the cumulative sum is visible at the time of packet loss.

Table 3 shows an overview of the queues that are related to our configuration. Queues that are not related to the ADSd orderbooks, and queues without detected non-zero fill levels have been omitted. Two observations can be made in this table: First, the sum of events in the queue Market→Orderbook: WorkerThread/1 is less than the amount of detected non-zero fill levels. A further analysis of this revealed that all the negative events seen in Figure 13 originate from this queue. The second thing that can be observed is that the Orderbook→Strategy queue did not appear due to the fact that no non-zero fill level was measured.

The observations made here are similar for a replay delay of 50μs (see Appendix D.2 for the corresponding table and graphs, and for a comparison of the cumulative sum of fill levels between 40μs and 50μs replay speed).

1:10 Orderbook-Strategy Configuration Since packet loss occurred without visible indication of a full queue, a different strategy configuration was tested, using a *1:10* orderbook-pair to strategy relation: In this configuration, the orderbook pair for the *ADSd* (Adidas AG) provides events to 10 strategies. In this configuration, the *ADSd* Orderbooks have to generate 10 times as many events as in the previous configuration, thus increasing the load on the orderbook thread, which should in turn be visible in the incoming queue for this orderbook.

The changed configuration resulted in the requirement to further decrease the replay-speed, as tests with a replay delay of 50μs failed because of lost packets at the feed decoders. A successful test run has been performed with a replay delay of 90μs.

A plot for a configuration where packet loss occurred, using a replay delay of 80μs, is shown in Figure 15. The most significant observation that can be made from the graph is the large peak at about 1/3 of the test runtime. An analysis done of this peak revealed that in fact the pattern was caused due to pressure on the Orderbook. Table 4 shows the fill levels over consecutive sample periods. It can be seen that a high fill state in the Feed:Chi-X→Market queue causes raised fill levels in the queues of the following stages.

At the time of packet loss, however, no significantly increased fill level can be detected. A complete overview of the related non-zero fill levels is shown in

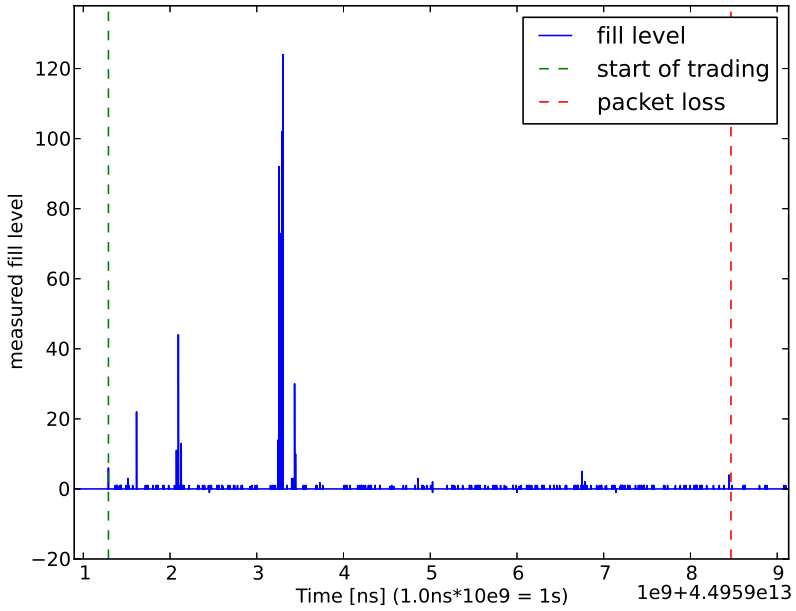

Figure 15: Queue fill levels at 80μs replay delay

Feed:Chi-X→Market	14	92	10	0	0	0
Market→OB:WorkerThread/1	0	0	44	102	124	0
Market→OB:ADSd	0	1	22	20	14	0

Table 4: Queue fill level pattern at 80μs replay delay

Queue	count($C \geq 1$)	$\sum C$	inCount, total
Feed to Market			
Feed:BATS to Market	73	291	26534
Feed:Chi-X to Market	75	296	22568
Market to Orderbook			
to OB:Chi-X:ADSd	6	84	353
to OB:BATS:ADSd	4	4	348
to OB:WorkerThread/1	28	372	6925
to OB:WorkerThread/0	35	65	11287
Orderbook to Strategy			
Chi-X:ADSd to S:ADSd-9	1	1	266

Table 5: Summary of non-zero fill levels at 80μs replay delay

Table 5. In comparison to Table 3, the sum of fill levels compared to the amount of non-zero fill levels detected is significantly larger, although as shown in the graph, this can be attributed mostly to single, large peaks.

Figure 16 shows the detected fill levels at a replay delay of 90μs. Note that the run-time of this test run is about 10 times longer than the previously analyzed run as indicated by the scaling factor in the bottom right corner of the graph, due to the fact that no packet loss occurred. The longer run-time leads to the visibility of a larger amount of peaks. The peaks appear mostly in the Feed→Market and the Market→Orderbook queues. An analysis of the peaks showed no significant patterns other than slight variations of the pattern described above. The two largest peaks originate from a loaded Orderbook:WorkerThread queue, which is at a fill-level of 50 and zero in the following sample period (at about 5.3 on the x-axis). The second large peak (near the 8-mark on the x-axis) is caused by both Feed→Market queues, with values of 63 (BATS) and 68 (Chi-X). Both fill levels are zero in the following sample period.

Simulation of Bursts As it was not possible to find any real indicator for a loaded queue or backpressure, it was tried to get closer to the production conditions by simulating the microburst load situation, with the constraints described in Section 3.1.

Again, the change of configuration enforced an increase of the replay delay. Starting off at bursts of 20 packets every 1 millisecond (50μs replay delay on average), the burst size was gradually decreased, until a configuration that did not cause packet loss was found at a burst size of 8 packets every millisecond. These values result in an average packet delay of 125μs, which again is a lower on-average replay delay than the value used in the previous tests. The resulting graph is shown in Figure 17. It can be seen that the detected non-zero fill levels are now generally larger, with the majority of fill levels being at 5 or less. Table 6 shows that these can largely be attributed to the Feed→Market queues, showing an average non-zero fill level of 2.29, whereas the Market→Orderbook queues show an average non-zero fill level of 1.48. This leads to the conclusion that the market stage compensates for the peaks. While similar values can be observed in Table 5, the corresponding graph and analysis shows that these values are dominated by a few large peaks.

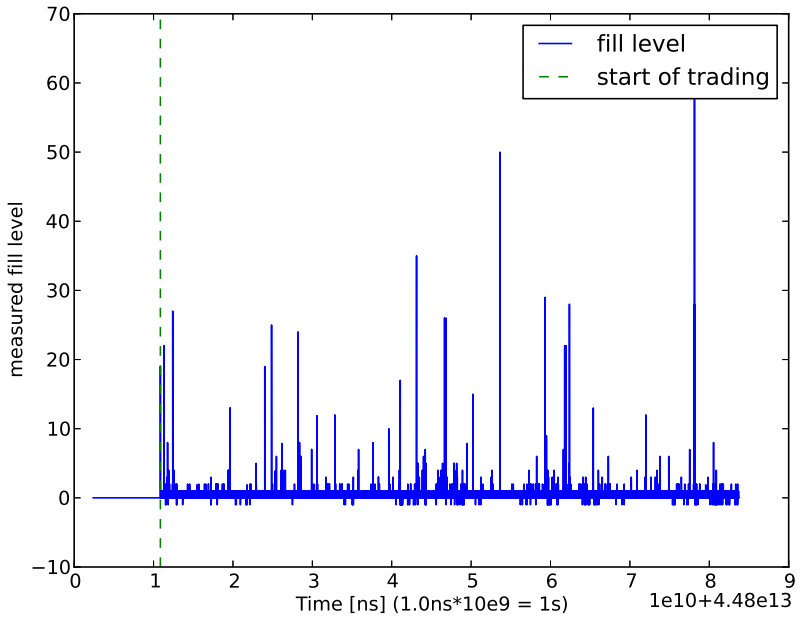

Figure 16: Queue fill levels at 90μs replay delay

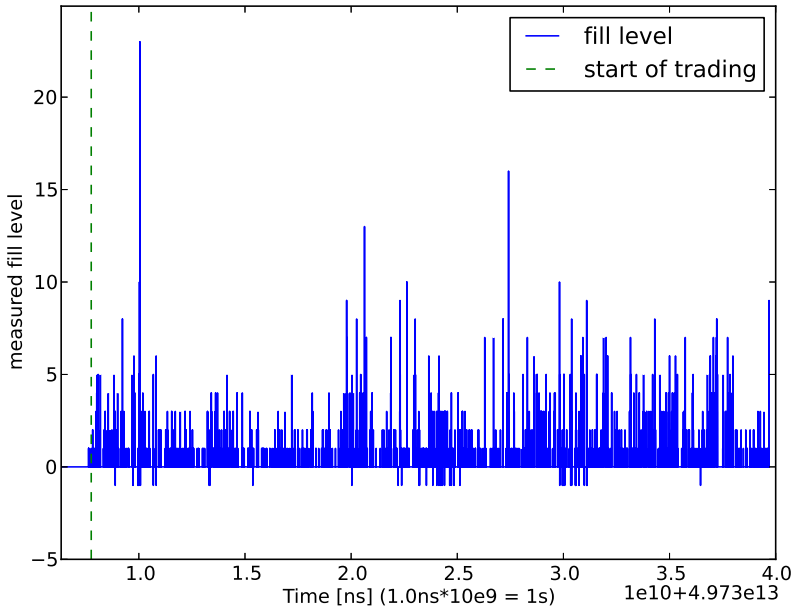

Figure 17: Queue fill levels at 1ms replay delay, bursts of 8 packets

Queue	count($C \geq 1$)	$\sum C$	inCount, total
Feed to Market			
Feed:BATS to Market	272	614	122807
Feed:Chi-X to Market	401	926	90088
Market to Orderbook			
to OB:Chi-X:ADSd	10	14	693
to OB:BATS:ADSd	28	42	2050
to OB:WorkerThread/1	107	153	35425
to OB:WorkerThread/0	208	312	54817
Orderbook to Strategy			
BATS:ADSd to S:ADSd-8	1	1	1287
BATS:ADSd to S:ADSd-4	1	1	613

Table 6: Summary of non-zero fill levels at 1ms replay delay, bursts of 8 packets

3.6 Results

In this section, the results gained from the measurements will be discussed. A discussion of the accuracy of the gained data along with the problems that appeared in testing will be done in Section 3.6.1. In Section 3.6.2, the suitability of the queue sampling method for performance analysis is discussed.

3.6.1 Quality of the Data

This section discusses whether the raw data generated through the queue sampling mechanism can be considered accurate. While there were no indications that an actual error was made, indicators for timing and scheduling issues were present that caused discrepancies between the data obtained through queue sampling and the data that could be gained through the tracing mechanism.

First of all, there was no indication that incorrect data was retrieved from the queues:

- Increasing the replay speed of the feed also caused a higher amount of non-zero fill levels to be detected. This fact is visualized in Figure 21. Using a burst input method caused higher peaks to appear — but those were almost never higher than the burst size.
- It could be observed that queues run full when shutting down the ITM system, where the fill level increased up to the point where no further packets could be added to the queue.[15]
- The data gathered from the RPC publish-subscriber queue clearly shows the pattern that was expected because of the inner workings of the publish-subscribe mechanism.

While it can therefore be assumed that the raw data collected from the queues is correct, a few *anomalies* were visible as well, which raise the questions whether the queue sampling method reflects the actual state of the queues at sampling:

- In Section 3.2, in some single instances, discrepancies were detected between the sampled fill level and the level calculated through the tracing mechanism. These could be found in about 0.5% of all cases. However, in these cases the duration that the element stayed in the queue was longer than the average duration an element stayed in this queue over the whole test case, strongly suggesting that a timing or scheduler problem is responsible for this discrepancy.
- The time required for sampling was unusually high. While some theories regarding caching and prefetcher issues could partially be applied to this problem in Section 3.4, both can not completely explain the long duration. Also, at the end of the mentioned section, a pattern in the sample duration has been found where one queue would take unusually little time to be sampled, and the following queue would account for the maximum sampled value. No valid explanation for this behavior could be found.

[15] These values had to be omitted from the graphs and tables, as the values tended to become very large and did not have any significant informative value.

Another problem with the data is that it was measured using test conditions that proved not to accurately reflect real-world conditions. This was displayed by the fact that in production, queues reach their maximum capacity in some situations, causing error messages and backpressure. This situation was never observed in testing. Instead, packets have been dropped at feed level, without any sign that this drop has been caused by backpressure from later stages. Primary differences between the test environment and the production environment, the missing *recovery* functionality and that the replay-tool is not able to simulate micro bursts correctly, have been discussed in Section 3.1. In addition to this, the actual strategy- and orderbook configuration could differ. Real-life strategies can be significantly more complex than the *SmartQuota* strategy used here and will access more orderbooks than shown here.

3.6.2 Suitability for Performance Analysis

The following will give a discussion of whether the queue sample method can be used for measurement and analysis of the given system, under the constraints listed above.

Indication of Imminent Failures Throughout all test runs, no indication of imminent packet loss could be found from looking at the queue sample data alone. It was also not possible to observe large fill levels through the queue sample data. The largest fill level recorded is shown in Figure 15, at a peak value of 124. Considering that the queues within the ITM are designed to hold 2048 events or more, and that the peak was not followed by any other significant non-zero fill levels, this value can not be considered to be a direct indicator for an imminent failure.

The change of configuration from a 1:1 to a 1:10 Orderbook to Strategy mapping however resulted in the requirement of significantly lowering the replay speed to avoid packet loss. The change in configuration did not introduce or remove a significant change in CPU load, as the number of threads and queues stayed the same. This indicates that some sort of backpressure has to be responsible for this, but this could be seen neither in the queues nor in the log files.

Relative Performance of the Stages One of the main goals of this thesis was to determine whether the queue sampling method can be used as a live-overview of the whole system, in which stages with relatively low performance can be identified.

Two main observations made in Section 3.5 point to two performance problems: high peak values and differences in relative fill levels between stages.

High peak values are visible, for example, in Figure 16. While these peaks are not an indicator for constant relative bad performance of a stage, they show that there is bad performance certain situations. Two possible causes for the peaks are listed below:

- The stage has to handle an uncommon special case that results in a larger amount of required processing.
- The scheduler interrupts the thread handling the stage, causing event processing to be stalled.

A special case of a peak value can be seen in Figure 15. Here, a high peak value in the Feed→Market queues leads to a load increase in both the Orderbook-Event queues (for example, Market→Orderbook:ADSd) and the Market→Orderbook:WorkerThread queue, which is responsible for scheduling the processing of the Events.

The second observation pointing to a possible performance problem is the differences in fill levels between the incoming queues of stages connected directly with each other. This observation is described in Table 6 and the corresponding analysis. It can be seen that events accumulate in the incoming queue of the market rather than in the incoming queues of the orderbooks, suggesting that the market is

1. slower in processing events than the feed decoders are, as the feed decoders are able to insert events into the queue faster than the market is able to remove them, and

2. slower in processing events than the orderbooks are, as the average fill level is higher in the market and the orderbook.[16]

It can be summarized that the changed configurations yield the type of result that was expected, but that, considering that replay speeds were used close to speeds where packet loss occurred were used, the fill levels and the amount of non-zero fill levels seen in testing are significantly lower than expected.

[16] Although the events are distributed to multiple orderbooks, there are two worker threads handling those orderbooks, and two market threads generating the events.

4 Conclusions and Future Work

This thesis describes the design, implementation, and validation of a system designed for performance analysis of a SEDA-based system by periodically measuring the inter-thread communication queues is described. It has been shown that, due to the low-latency characteristics of the system, it is not possible to directly derive performance-related statistics from the sampled data in this specific implementation.

A basic implementation approach has been given, along with implementation alternatives for the topology analysis and with suggestions for improvements for the sampling code and the generation of RPC events. The design was required to be less intrusive than the currently implemented *tracing* framework, and has met this requirement, as changes in the code using the queues were only required where the queues were instantiated, to add topology information to the queue.

When testing various aspects of the queue, it was shown that a performance impact can be measured at very high sample frequencies. Further optimizations to the sampling mechanism, as described in the *Implementation* section, could help to mitigate the problem. When testing whether performance issues were visible, test cases had to be specifically crafted to generate data that could be used as indicators for performance problems. The test cases yielded the expected results only partially, as it was not possible to find filled queues even when performing the test at replay speeds close to packet loss. It was, however, possible to detect indicators for performance problems, such as peaks and load patterns.

The remainder of this section states ideas for future work based on the work done in this thesis, split into work that would focus on implementation details and into improvements regarding testing and simulation of the approach.

Design and Implementation The fixed-rate sample mechanism proved to cause problems at high rates, but a higher sample frequency would be desirable to obtain more accurate results. The use of an adaptive sampling method, where queues with higher activity would be sampled more frequently, has been proposed in section 2.3. The implementation, however, requires strategies that are able to derive a good sampling frequency from the data available.

The mechanism used to derive the topology information can be used for thread pools, shared threads, and combinations of those. In this thesis, however, deriving this information from the data required additional knowledge of the system layout and configuration. It could be desirable to refine the topology mechanism used, in order to automatically generate a complete description of the connections between threads and queues.

The RPC mechanism used proved to restrict the sampling frequency. In Section 2.4, approaches for improvements are given, such as asynchronous event generation, event coalescing, and event filtering. These approaches will have to be evaluated in order to support higher sample frequencies.

As discussed in Section 2, the queue sampling method can be extended to be used for inter-process communication and memory pool queues. The latter would require a different form of analysis and would have to resolve conflicts concerning the singleton `QueueRegistry` and possibly solve problems regarding the determination of the in-counter and out-counters in a multi-process environment.

Validation Techniques for increasing the visibility of performance isssues are highly desirable. As previously discussed, the conditions in the test environment did not allow to reproduce the problems that appeared in production. An important future task is to determine the parameters that resulted in the different behavior that was observed while testing. It could also be possible to introduce latencies in the stage implementation to artificially generate backpressure and higher non-zero fill levels.

The description of the arrival process and the service process at the incoming queues can be done in a more formal way. This would allow to better determine what system properties lead to data, where performance related information can be extracted. On the other hand, having a formal model of the system may help to interpret the queue fill states and patterns in the queue fill states. Simulation frameworks exist that allow modeling of queueing theory processes such as SimPy,[17] and frameworks that allow discrete event simulation such as Chute,[18] possibly allowing for better inspection of the underlying causes of fill states and fill state patterns.

A cause for the high duration of the queue sampling described in Section 3.4 could not be found, and requires more investigation. Although caching issues have largely been ruled out for this specific problem and the performance impact of the sampling code on the ITM does not give conclusive results because of the high standard deviation, caching issues have an impact on latency-critical applications. This is, for example, displayed by the fact that others [23] make large efforts to minimize cache misses. A more in-depth analysis of these effects in a SEDA application could be an interesting research topic.

[17] `http://simpy.sourceforge.net/`, accessed 13-July-2012
[18] `http://code.google.com/p/chute/`, accessed 13-July-2012

5 References

[1] S. Adve, G. Agha, M. Frank, M. Garzarán, J. Hart, et al. Parallel computing research at Illinois: The UPCRC agenda. Online; URL `http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf`, 2008, accessed 24-June-2012.

[2] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[3] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 International Conference on Management of Data, SIGMOD*, volume 11, pages 1071–1080, 2011.

[4] H. Brunst and B. Mohr. Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG. *OpenMP Shared Memory Parallel Programming*, pages 5–14, 2008.

[5] D. F. Carr. How Google Works. In Base Line Magazine; URL `http://www.baselinemag.com/c/a/Infrastructure/How-Google-Works-1`, 2006, accessed 18-June-2012.

[6] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2007.

[7] I. Chung, R. Walkup, H. Wen, and H. Yu. MPI performance analysis tools on Blue Gene/L. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 16–16. IEEE, 2006.

[8] J. Corbert. Fast Reader Locks. In Linux Weekly News; URL `http://lwn.net/Articles/21379/`, 2003, accessed 24-June-2012.

[9] R. Cox. Bell labs and CSP threads. Online; URL `http://swtch.com/~rsc/thread/`, 2011, accessed 26-June-2012.

[10] R. Cox. Profiling go programs. In The Go Programming Language Blog; `http://blog.golang.org/2011/06/profiling-go-programs.html`, 2011, accessed 26-June-2012.

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

[12] U. Drepper. What every programmer should know about memory. Online; URL `http://www.akkadia.org/drepper/cpumemory.pdf`, 2007, accessed 12-July-2012.

[13] M. Durbin. *All About High-Frequency Trading*. All About Series. McGraw-Hill, 2010.

[14] M. P. I. Forum. Mpi: A message-passing interface standard, version 2.2. Online; URL `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`, 2009, accessed 12-July-2012.

[15] M. Geimer, F. Wolf, B. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[16] A. Hanif. Colocation and latency optimization. *UCL Department of Computer Science: Research Notes*, 12(04), 2012.

[17] Intel® Corporation. Enhanced Intel SpeedStep® Technology for the Intel® Pentium® M Processor, 2004.

[18] Intel® Corporation. Intel® threading building blocks for open source. `http://threadingbuildingblocks.org/`, 2012, accessed 26-June-2012.

[19] D. J. Leinweber. *Nerds on Wall Street: Math, Machines and Wired Markets*. Wiley, 1 edition, 2009.

[20] Z. Li, D. Levy, S. Chen, and J. Zic. Auto-tune design and evaluation on staged event-driven architecture. In *Proceedings of the 1st Workshop on Model Driven Development for Middleware (MODDM'06)*, pages 1–6. ACM, 2006.

[21] S. Mirtaheri, E. Khaneghah, and M. Sharifi. A case for kernel level implementation of inter process communication mechanisms. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–7. IEEE, 2008.

[22] M. Mohsen, R. Abdullah, and Y. Teo. A survey on performance tools for openmp. Online; URL `http://www.waset.org/journals/waset/v49/v49-135.pdf`, 2009, accessed 22-June-2012.

[23] M. Thompson, D. Farley, et al. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. Online; URL `http://disruptor.googlecode.com/files/Disruptor-1.0.pdf`, 2011, accessed 6-June-2012.

[24] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth Symposium on Operating System Principles*, pages 230–243, 2001.

[25] K. Wright and K. Kang. Performance analysis of various mechanisms for inter-process communication. Online; URL `http://osnet.cs.binghamton.edu/publications/TR-20070820.pdf`, 2007, accessed 22-June-2012.

[26] Zero-C Inc. The ICE RPC framework. Online; `http://www.zeroc.com/overview.html`, 2011, accessed 26-June-2012.

A Source: Queue Sample Thread

This code describes the function performing the queue sampling. It is called by the thread's `run` function in a configurable interval (queue sample period).

```
1  int64_t StatisticsProvider::getQueueSampleData()
2  {
3   uint64_t count = 0;
4   uint64_t incount = 0;
5   QueueRegistry *reg = QueueRegistry::getInstance();
6
7   ScopedBlockingLock sampleLock(reg->registryLock);
8
9   std::map<SampleableQueue *, QueueMeta> queues = reg->
        getQueues();
10
11  // send topology updates first.
12  if (reg->getClearTopologyUpdated()) {
13   TopologyChangedEvent *topoEv = getCurrentTopologyLocked
         (queues);
14   publishEvent(topoEv);
15  }
16
17  std::vector<QueueSampleData> vQsd;
18  for (std::map<SampleableQueue *, QueueMeta>::iterator it
        = queues.begin();
19       it != queues.end(); ++it) {
20    QueueSampleData qsd;
21    qsd.queueId = (size_t)it->first;
22    qsd.inCount = it->first->incount();
23    qsd.outCount = it->first->outcount();
24    qsd.timestamp = nsSinceMidnight();
25
26    vQsd.push_back(qsd);
27  }
28
29  QueueStateSampleEvent * sdev = new QueueStateSampleEvent
        (nsSinceMidnight(), vQsd);
30  publishEvent(sdev);
31  return 0;
32 }
```

B Source: Discrepancy Detection Tool

This script reads in both queue sample data and trace data. It then generates an output for every time the fill state derived from the sampled data does not match the fill state that is implied by the trace data.

Input: Two ASCII files. The first contains in every line a pair of timestamps corresponding to the insertion and removal time of the queue that is analyzed. These two timestamps should be separated by a comma (","). The second one corresponds to the filtered *dump* of the Python data structure. It consists of 4 key/value pairs separated by "=", where the keys are "timestamp", "inCount", "outCount" and "queueId". The dump should be filtered, so that the value of "queueId" is constant throughout the file. The 4 values should then be terminated by a line containing only "–".

Output: For each time a trace has not been "detected" by the queue sample mechanism at the sample timestamp T_{sample}, the script outputs

1. A line containing only the insertion timestamp of the trace
2. A line containing the removal timestamp of the trace, followed by " = " and the calculated duration the corresponding event spent in the queue.

if at least one trace has not been detected, a line is outputted containing T_{sample}, the fill state detected by the sample mechanism and the fill state detected using the tracing mechanism, separated by single spaces.

```
1  #include <iostream>
2  #include <map>
3  #include <vector>
4  #include <string>
5  #include <fstream>
6  #include <boost/algorithm/string.hpp>
7  #include <boost/lexical_cast.hpp>
8  #include <map>
9  #include <algorithm>
10
11
12 using namespace std;
13 using namespace boost;
14
15 // in each pair: timestamp, queueid
16 typedef vector<pair<uint64_t, int64_t> > sampledata_t;
17 // in each pair: seconds, nanoseconds
18 typedef vector<pair<uint64_t, uint64_t> > tracedata_t;
19 bool load_sampledata(char * filename, sampledata_t &
       sampledata) {
20   string line;
21   uint64_t inCount, outCount, timestamp;
22   ifstream queuedata(filename);
23   while (!getline(queuedata, line).eof()) {
24     trim(line);
25     if (line == string("--")) {
```

```
26        sampledata.push_back(make_pair(timestamp, inCount-
              outCount));
27        //the following line can be used for plot
              generation:
28        //cout << timestamp << "," << inCount << "," <<
              outCount << endl;
29        continue;
30      }
31      vector<string> tmp;
32      split(tmp, line, is_any_of("␣=\n"), token_compress_on
            );
33      if (tmp.size() != 2) {
34        //cout << line.size();
35        cerr << "Malformed␣line:␣'" << line << "',␣" << tmp
              .size() << endl;
36        return -1;
37      }
38      if (tmp[0] == "timestamp") {
39        timestamp = lexical_cast<uint64_t>(tmp[1]);
40      } else if (tmp[0] == "inCount") {
41        inCount = lexical_cast<uint64_t>(tmp[1]);
42      } else if (tmp[0] == "outCount") {
43        outCount = lexical_cast<uint64_t>(tmp[1]);
44      } else if (tmp[0] == "queueId") {
45        continue;
46      } else {
47        cerr << "Invalid␣Token:␣" << tmp[0] << "␣in␣"  <<
              line << endl;
48      }
49    }
50  }
51
52  int main(int argc, char ** argv) {
53    if (argc != 3) {
54      cerr << "Usage:␣" << argv[0] << "␣<tracetable.txt>␣<
            queuesampledata.txt>" << endl;
55      return 1;
56    }
57
58    ifstream tracetbl(argv[1]);
59
60    tracedata_t tracedata;
61    string line;
62    int tracedata_count = 0;
63    uint64_t tracedatadiff_sum = 0;
64    while (!getline(tracetbl, line).eof()) {
65      vector<string> tmp;
66      split(tmp, line, is_any_of(","), token_compress_on);
67      uint64_t start = lexical_cast<uint64_t>(tmp[0]);
68      uint64_t stop = lexical_cast<uint64_t>(tmp[1]);
```

```
69      tracedata.push_back(make_pair(start, stop));
70      tracedata_count++;
71      tracedatadiff_sum += stop-start;
72    }
73
74    sort(tracedata.begin(), tracedata.end());
75    cerr << "Tracedata loading completed." << endl;
76
77    sampledata_t sampledata;
78    load_sampledata(argv[2], sampledata);
79
80    // All data has been loaded, start the analysis
81    int i = 0;
82    int ec = 0;
83    int ec_sumtslen = 0;
84    for (sampledata_t::iterator it = sampledata.begin(); it
          != sampledata.end(); ++it) {
85      i = 0;
86      tracedata_t::iterator td =
87        lower_bound(tracedata.begin(), tracedata.end(),
              make_pair(it->first, (uint64_t)0));
88
89      if (td == tracedata.begin() || td == tracedata.end())
90        continue;
91      --td;
92
93      // "slide" window, and output all matching traces
94      while (td != tracedata.begin() && td != tracedata.end
          () && td->first < it->first &&
95          td->second > it->first) {
96        cout << td->first << endl << td->second << " = " <<
              td->second - td->first << endl;
97        ec_sumtslen += (td->second - td->first);
98        ++i; --td;
99      }
100
101     if (it->second != i) {
102       cout << it->first << " " << it->second << " " << i
              << endl;
103       ec++;
104     }
105   }
106   cout << "Of " << tracedata.size() << " traces, " << ec
        << "failed. Average tracelength: "
107       << (double)tracedatadiff_sum/tracedata.size()
108       << ", average tracelength of failed traces:"
109       << ec_sumtslen<< endl;
110 }
```

C Data Preparation Process

The following is a description of the process used to generate various forms of input data for further analysis.

C.1 Trace Data

The trace data is stored in a *postgresql* database, as an array of integers for each row. In order to have a portable form of the data without requiring to have a database server installed on the system where the analysis is performed, a text dump has been created from the database using

```
1 psql -c "SELECT * FROM feed_trace WHERE session = 100;" >
    ADSd.tracetable
```

where 100 is the ID of the trace corresponding to the test run and strategy observed. ADSd is the name of the instrument that the strategy observes. Further filtering removes everything except the trace data from the database table dump:

```
1 sed -e 's/.*{//' -e s/}// ADSd.tracetable > ADSd.stripped
    .tracetable
```

Removal of the first and the last lines, which only contain table header and summarizing statistics information, can be done by hand. The file now contains, in each line, the data trace (as a comma-separated-values (CSV) table). The resulting data set can then be brought into a form that is suitable for usage with the tool described in appendix B: Fields 8 and 9, corresponding to the insertion and extraction timestamp of the *Orderbook-to-strategy*-queue are selected from the table.

```
1 cut -d, -f8,9 ADSd.stripped.tracetable > ADSd-OB-STRAT.
    tracetable
```

To generate the raw data for Figure 10, one can append the line number after a single value (in this example, the insertion timestamp):

```
1 cut -d, -f1 ADSd-OB-STRAT.tracetable | awk '{ print $1" "
    NR }' > ADSd-OB-STRAT.event-increment
```

C.2 Sampled Data

Conversion from ICE data For the most part, the Python module `pickle` and the alternative, C-based version `cPickle` has been used to store the raw data as received by the Python ICE client. Two problems with this module should be noted:

- The module stores class-related information, and requires these classes to be loadable when reversing the serialization process. I did run into the problem, that although the representation[19] of the data dump did only show a standard python data structure, de-serializing of this data led to errors. As pickle stores class information, both the *slice2py* generated

[19] as reported by the objects __repr__ method in python

module and the ICE libraries had to be present. The ICE structures would not load, apparently due to incompatibilities between the ICE internal data structures on Linux and Mac OS X.

- Neither pickle modules does support stream-based reading and writing. Therefore, loading and storing the data required up to 20 times the memory of the resulting, "pickled" file. In some instances the resulting files size turned out to be as large as 150MB, and the process of saving and loading this file caused the system running the process to swap.

Both problems can partially be solved by using the `json` module for serialization instead of `pickle`. `json` will still raise a TypeError when trying to serialize a non-builtin type, but it is possible to specify a converter function in which the "unknown" objects can be parsed:

```
1  def serialize_helper(obj):
2    o = None
3    try:
4      # If the object is a SingleQueueSampleEvent, extract
5      # corresponding data
6      o = { 'inCount':    obj.inCount,
7            'outCount':   obj.outCount,
8            'timestamp':  obj.timestamp,
9            'queueId':    obj.queueId }
10   except:
11     pass
12   else:
13     # if the object is a QueueSampleEvent, extract
14     # sampledata and timestamp
15     o = { 'sampledata': obj.sd,
16           'timestamp':  obj.timestamp }
17   except:
18     pass
19   if o == None:
20     raise ValueError("Object type not supported: " + repr
         (obj))
21   return o
22
23 json.dump(data, file, default=serialize_helper)
```

Conversion from Python Data The raw data generated by the queue sample mechanism can be formatted using the *pprint* in such a way that it is correctly indented and values appear line-by-line. This can make further processing more easy.

Alternatively, one can use the representation generated by the ICE objects for this. This format will result in an output that looks similar to this:

```
1  [{
2    timestamp = 99999990000
3    sd = {
4      [0] = {
```

```
5           inCount = 3999
6           outCount = 4000
7           queueId = 1
8           timestamp = 99999999998
9         }
10        [1] = {
11          ...
12          queueId = 2
13          ...
14        }
15    }
16  }
17  {
18    timestamp = 99999999999
19    sd = {
20        ....
21    }
22  }]
```

Such a format can easily be filtered using standard UNIX utlities. For example, reducing the file to only the samples for a specific queue, in this example Queue 1, can be done using *grep*:

```
1  grep -B 2 -A 1 -E 'queueId = 1$' sampledata.out
```

This will include the two lines before and one line after the match to the output, separating the output by a line containing only "–". Manually inserting this line to end of the output makes it suitable for usage in the tool described in Appendix B.

C.3 Plot Generation

Plots have been generated using the *matplotlib* framework. This example shows the method used to generate the plot in Figure 10. For this, a cumulative count of traces at a given timestamp is required. This can be done using line numbers, as shown in the *Trace data* section of this appendix. The example code below expects a file containing, for each trace, a line containing a sequentially increasing number (the cumulative count of traces, which can be created using line numbers) and a timestamp corresponding to the insertion timestamp. In the example code below, this file is named `ADSd-OB-STRAT.inCounts.tracetable.pluscount`.

Preparation of the queue sample data can be done by removing the trace-related code from the program shown in appendix B, and outputting the combined inCount values of both observed queues along with the timestamp of the queue that was sampled first.

```
1  import matplotlib.pyplot as p
2
3  f = open("ADSd-OB-STRAT.inCounts.tracetable.pluscount")
4
5  x = []
6  y = []
7  x2 = []
```

```
 8 y2 = []
 9 for l in f:
10         l = l.rstrip().split(" ")
11         x.append(l[0])
12         y.append(l[1])
13
14 f = open("sampledata.combined.pluscount")
15 for l in f:
16         l = l.rstrip().split(" ")
17         x2.append(l[0])
18         # Remove 5 elements (number of control events)
19         y2.append(int(l[1])-5)
20
21 p.plot(x, y, drawstyle='steps-post', label='trace count')
22 p.plot(x2, y2, drawstyle='steps-post', label='sampled
       count')
23 p.legend()
24 p.show()
```

D Additional Graphs and Tables

D.1 Queue Sample Performance

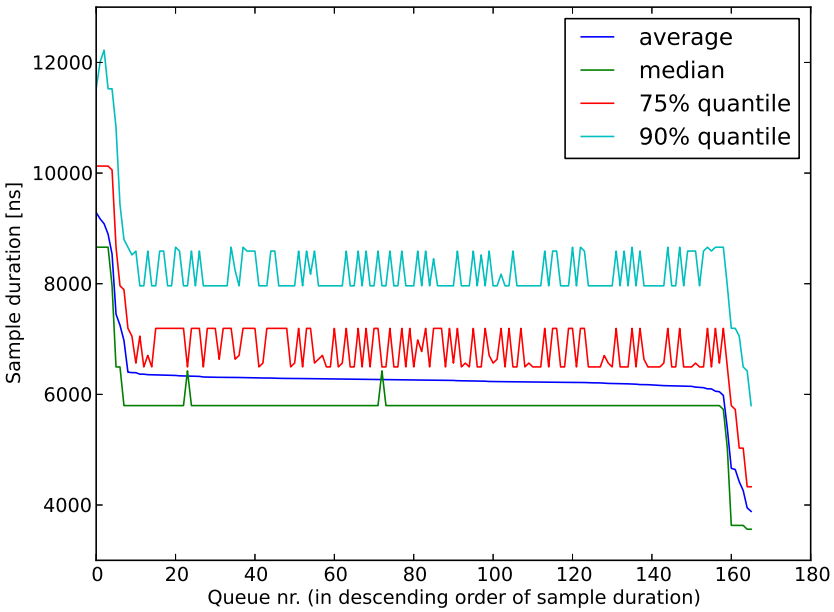

Figure 18: Queue sample performance at 90μs replay delay, quantiles

D.2 Queue Fill Level at 50μs Replay Delay

Queue	count($C \geq 1$)	$\sum C$	inCount, total
Feed to Market			
Feed:BATS to Market:	212	241	55714
Feed:Chi-X to Market:	125	182	42782
Market to Orderbook			
to OB:BATS:ADSd	3	3	625
to OB:WorkerThread/0	39	23	23846
to OB:WorkerThread/1	38	38	14758

Table 7: Summary of non-zero fill levels at 50μs replay delay

Legend: count($C \geq 1$) Number of samples with non-zero fill level
$\sum C$ Sum of (non-zero) fill levels

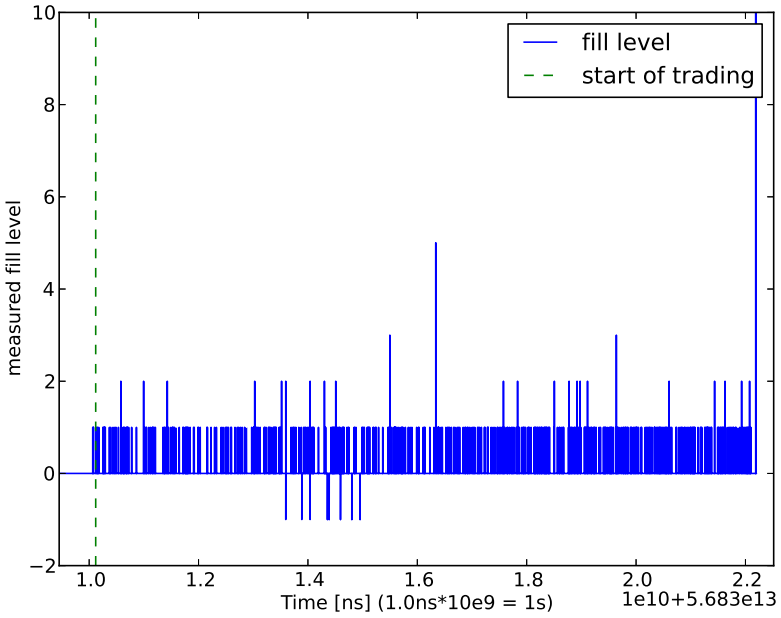

Figure 19: Queue fill levels at 50μs replay delay

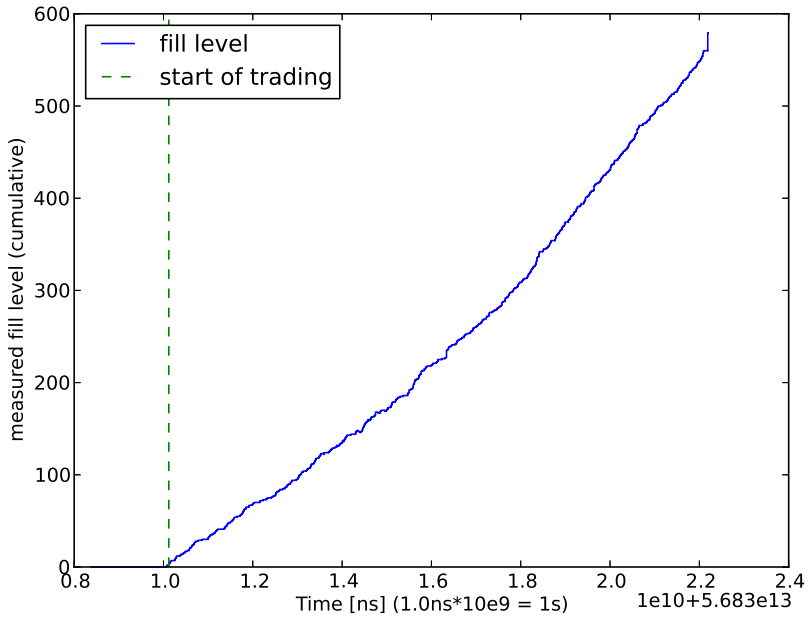

Figure 20: Queue fill levels at 50μs replay delay, cumulative sum

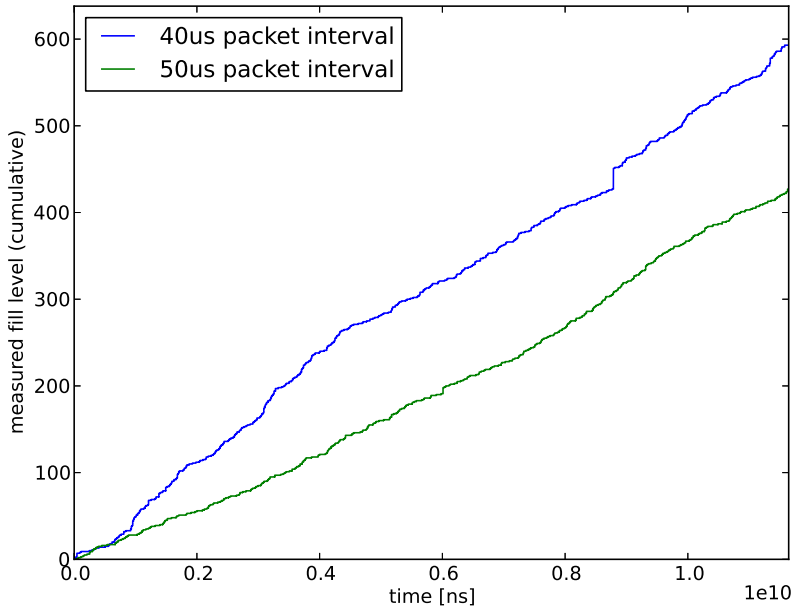

Figure 21: Comparison of cumulative non-zero fill level sum