Distributed Pipe-and-Filter Architectures with TeeTime

Master's Thesis

Florian Echternkamp

May 21, 2017

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 21. Mai 2017

Abstract

The Pipe-and-Filter architectural style provides easily exploitable parallelism of each filter stage. A single computer limits the parallel execution by its amount of CPU cores. To further exploit the parallelism the Pipe-and-Filter architecture can be executed distributively on multiple computer nodes.

In this thesis, we present an approach to implementing a distributed Pipe-and-Filter architecture with the TeeTime framework. Our implementation provides efficient distributed communication, which uses TCP, UDP, or is encrypted via SSL over TCP. Also, we provide an automatic remote deployment and remote execution, as well as fault handling. We successfully evaluate the feasibility and performance improvements of our implementation. In our example applications, we gain performance improvements of approximately 40%. Furthermore, we present a working DSL for distributed configurations to simplify the usage of distributed Pipe-and-Filter architectures.

Our implementation is part of the TeeTime framework, which is available in a version with and a version without our extension. We provide the DSL as an Eclipse plug-in.

Contents

Chapter 1

Introduction

1.1 Motivation

Pipe-and-Filter architectures are used for stepwise processing of data streams. Each processing step is represented by a filter module, which is connected to other filters via pipes. Due to their Filter-based modularization, Pipe-and-Filter architectures are suitable for parallelization. To optimize the performance, each filter can be executed in parallel. However, the parallelization on a single system is limited by the number of processor cores. When more filters than cores are used, the filters must share the cores and thus could not be executed parallel. Typical personal computer systems use between 2 and 4 Cores. To provide parallelization of filter quantity above the amount of these cores distributed computing can be utilized. The filters are distributed to multiple computers, named nodes, which communicate via a network. Consequently, it is important that the network overhead stays lower than the performance improvement of the parallel execution of the filters to gain overall performance enhancements.

A distributed Pipe-and-Filter architecture yields further benefits concerning data locality, which is necessary for processing Big Data with multiple sources. Instead of accessing the sources via the network the data is directly handled by filters on or nearby the nodes of data origin and can be joined later, due to the distributed architecture.

Distributed Pipe-and-Filter architectures are not supported by the Pipe-and-Filter Java Framework TeeTime [Wulf et al. 2014; 2016], which is developed at the Kiel University. The goal of this Master's Thesis is to extend TeeTime to support distributed Pipe-and-Filter architectures. The focus is on providing a feasible approach to implementing a distributed Pipe-and-Filter architecture with efficient distributed communication.

1.2 Goals

1.2.1 G1: Evaluation of Java Frameworks for Distributed Application Development

The first goal is to use an existing Java framework for distributed application development to implement the distributed Pipe-and-Filter architecture in TeeTime. Therefore we evaluate suitable Java frameworks. The following subgoals describe the main features the framework shall provide.

G1.1: Providing Efficient Distributed Communication

Currently, filters in TeeTime can be only connected by local pipes on the same node. Filters should get connected via the network with a minimal network communication overhead. Java objects need to be serialized for network transportation. The framework should provide distributed communication with efficient serialization and data transmission. Depending on whether it is more important to obtain high throughput or to ensure that no data get lost, the user should be able to choose between reliable TCP and faster, non-blocking UDP transmission.

G1.2: Providing Fault Tolerance

During execution, the application should be able to detect and log node and network failures. After a failure occurs, all nodes should be terminated.

G1.3: Providing Remote Deployment

The configuration parts should be automatically deployed to the nodes based on a single distributed TeeTime configuration.

G1.4 (optional): Providing Encrypted Data Transmission

Network traffic might be observable for other users. We supply optional encryption to protect the data transmission.

1.2.2 G2: Implementation of a Distributed Pipe-and-Filter Architecture

We implement the subgoals G1.1 up to G1.4 in TeeTime either based on the framework evaluated in G1 or by a custom solution. A single TeeTime configuration for all nodes is the basis for the distributed execution of the Pipe-and-Filter architecture.

1.2.3 G3 (optional): Adding Support for Distributed Pipes in the TeeTime DSL

TeeTime already provides a Domain-Specific Language (DSL) to define a configuration with pipes and filters. We extend this DSL to support distributed configurations.

1.2.4 G4: Evaluation of Our Approach

We evaluate the feasibility and performance of G2.

G4.1: Feasibility

The application is able to communicate (optionally encrypted) via the network. When a fault occurs, the application should detect and handle it.

G4.2: Performance

We test the performance of the implemented distributed Pipe-and-Filter architecture against a non-distributed Pipe-and-Filter application by performing performance tests on self-developed sample applications. We test TCP and UDP, and also measure the security overhead for the encryption.

1.3 Document Structure

We explain the required foundations and technologies, which we used for the implementation, in Chapter 2. Subsequently, we describe our evaluation of potential Java frameworks for the implementation in Chapter 3. Afterward we explain our developed approach of a distributed Pipe-and-Filter architecture in Chapter 4 and our approach for the extension of the TeeTime DSL in Chapter 5. Then we evaluate our implementation in Chapter 6. After that, we compare our approach to related work in Chapter 7. Finally, we present the conclusions and future work in Chapter 8.

Chapter 2

Foundations and Technologies

2.1 The Pipe-and-Filter Architectural Style

The Pipe-and-Filter architectural style [Sommerville 2012; Otero 2012; Frank Buschmann 2007] is a basic structure for data flow systems. The software is decomposed into data processing components named filters and one-directional data transferring components called pipes. The software is composed by connecting the filters via the pipes to a pipeline. Each filter incrementally processes the data stream. The incremental processing enables concurrent processing of the filters and thus a distributed Pipe-and-Filter architecture.

2.2 The Pipe-and-Filter Framework TeeTime

TeeTime [Wulf et al. 2014; 2016; 2017] is a Pipe-and-Filter framework for Java. It implements the generalized Tee-and-Join-Pipeline design pattern [Wulf et al. 2014], which allows filters (called stages in TeeTime) to be connected to more than one input and output pipe via input and output ports. The framework supports a fully type-safe environment through typed ports. TeeTime will be extended to implement distributed Pipe-and-Filter architectures.

2.3 Distributed Systems

Coulouris et. al. describes a distributed system as a system in which

> software components located at networked computers communicate and coordinate their actions only by passing messages.

Distributed systems can provide higher performance than a single system by sharing concurrent resources. Though additional sources of failures accrue. The network can fail and affect the data transmission, or a node can fail and harm the associated data processing steps. Those faults must be detected and handled appropriately [George F. Coulouris and Blair 2011].

2.4 Communication Patterns for Distributed Systems

The following communication patterns are quoted from the book *Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems* by J. Silcock et. al. [Silcock and Gościński 1995].

2.4.1 Message Passing

> Message passing is the basis of most interprocess communication in distributed systems. It is at the lowest level of abstraction and requires the application programmer to be able to identify the destination process, the message, the source process and the data types expected from these processes [Silcock and Gościński 1995, Chapter 2].

There are multiple variants of message passing available, e.g.:

Blocking/non-blocking The sender and receiver methods can be either executed blocking or non-blocking.

Buffered/unbuffered messages Messages can be sent directly to the receiving process, which must be running before the messages are sent to be able to receive them. Alternatively, the message can be directed to a message buffer, where the messages are saved until the receiving process is ready to receive them.

Reliable/Unreliable send Unreliable send once sends the message and does not consider it any longer. Reliable send requires an acknowledgment message and automatically retransmits a failed message.

2.4.2 Remote Procedure Calls

> Message passing leaves the programmer with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) relieves this burden by increasing the level of abstraction and providing semantics similar to a local procedure call [Silcock and Gościński 1995, Chapter 3].

2.4.3 Distributed Shared Memory

> Distributed shared memory is memory which, although distributed over a network of autonomous computers, gives the appearance of being centralized. The memory is accessed through virtual addresses, thus processes are able to communicate by reading and modifying data which are directly addressable. [..] The operating system has to send messages between machines with requests for memory not available locally and to make replicated memory consistent [Silcock and Gościński 1995, Chapter 4].

2.5 Fault Tolerance Design Patterns

The following fault tolerance patterns are for the most part cited from the book *Patterns for Fault Tolerant Software* by Robert Hammer [Hanmer 2013]. The patterns are grouped by Architectural Patterns (Section 2.5.1), Detection Patterns (Section 2.5.2) and Error Recovery Patterns (Section 2.5.3).

2.5.1 Architectural Patterns

Redundancy

> Provide redundant capabilities that support quick activation to enable error processing to continue in parallel with normal execution [Hanmer 2013, Chapter 4.3].

There are multiple redundancy variants:

Active/Active provide for each unit a whole redundant one, which are both active and load sharing.

Active/Standby also provide a redundant unit for each unit, but which is idle as long as the primary unit is performing well.

N+M Redundancy provides only M redundant standby units, with M<N.

Minimize Human Interaction

> Design the system in a way that it is able to process and resolve errors automatically, before they become failures. This speeds error recovery and reduces the risk of procedural errors contributing to system unavailability [Hanmer 2013, Chapter 4.5].

Someone in Charge

> All fault tolerant related activities have some component of the system ('someone') that is clearly in charge and that has the ability to determine correct completion and the responsibility to take action if it does not complete correctly. If a failure occurs, this component will be sure that the new failure doesn't stop the system [Hanmer 2013, Chapter 4.8].

Fault Observer

> Report all errors to the fault observer. The fault observer will ensure that all interested parties receive information about the errors that are occurring [Hanmer 2013, Chapter 4.10].

The system does not stop when errors are detected; it automatically corrects them. With the fault observer, users will know what faults and errors have been detected and processed.

2.5.2 Detection Patterns

System Monitor

Create a Monitor to study system behavior or the behavior of specific parts of the system to make sure that they continue operating correctly. When the watched components stop, the monitor should report the occurrence to the Fault observer and initiate corrective action [Hanmer 2013, Chapter 5.4].

Heartbeat

The System monitor should see a periodic heartbeat from the monitored task, see Figure 2.1. If the monitored task does not supply a heartbeat response within the required time then recovery action should be taken [Hanmer 2013, Chapter 5.5].

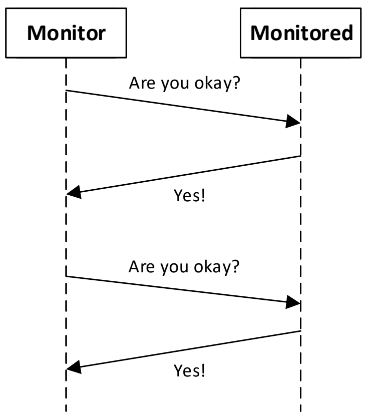


Figure 2.1. Regularly scheduled heartbeats (based on [Hanmer 2013, Chapter 5.5])

Acknowledgment

Send an acknowledgment for all requests, see Figure 2.2. All requests should require a reply to acknowledge receipt and to indicate that the monitored system is alive and able to adhere to the protocol. If the acknowledgment reply is not received then a failure should be reported to the Fault observer and error processing should be initiated [Hanmer 2013, Chapter 5.6].

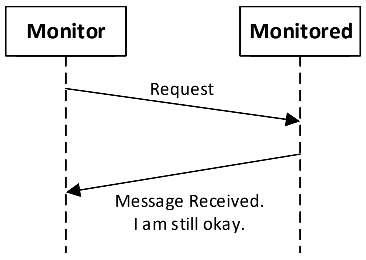


Figure 2.2. Requests are acknowledged (based on [Hanmer 2013, Chapter 5.6])

Watchdog

Add in the capability for the monitor to observe the monitored task's activities, much as a Watchdog tends the flock. This Watchdog can be either a hardware or a software component depending on the system requirements, but in either case it will watch visible effects of the monitored task. The monitored task will not be modified, see Figure 2.3. The Watchdog should take some actions to get the monitored task back into the flock if it strays too far from expected and desired behavior [Hanmer 2013, Chapter 5.7].

Realistic Thresholds

Set the messaging latency based upon the worst case communications time combined with the time required to process one Heartbeat message. Set the detection latency based upon the criticality of the functionality. Make it a multiple of the messaging latency. Use a smaller multiple for extremely critical or unique tasks, larger for redundant tasks. Set the latencies so that the monitor will be informed in a timely enough manner to meet the availability requirement, and yet is the maximum possible so that false triggers don't occur [Hanmer 2013, Chapter 5.8].

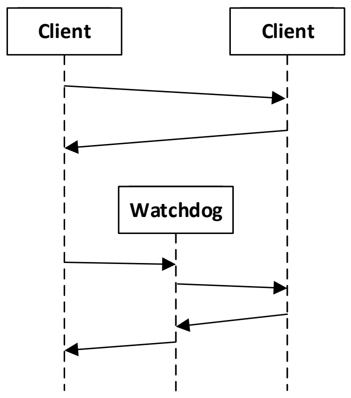


Figure 2.3. A Watchdog interposes itself on message traffic (based on [Hanmer 2013, Chapter 5.7])

2.5.3 Error Recovery Patterns

Error Handler

> Separate error processing code in special handling blocks for easier maintenance and to facilitate new handlers being added in the future [Hanmer 2013, Chapter 6.3].

Resume execution

Restart Resume execution by restarting the program from the beginning [Hanmer 2013, Chapter 6.4].

Rollback Resume normal execution by moving to a state in the execution path but before the error occurred [Hanmer 2013, Chapter 6.5].

Roll-Forward Resume normal execution by advancing to a future state that would have been reached if the error had not occurred [Hanmer 2013, Chapter 6.6].

Return to Reference Point Resume execution by returning to a specific known state. That place might not have been in the execution path that led to the error, but it is known to be safe [Hanmer 2013, Chapter 6.7].

Failover

> Recover by switching to a redundant unit, see Figure 2.4 [Hanmer 2013, Chapter 6.9].

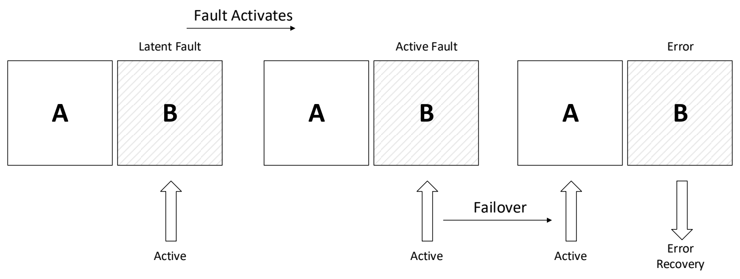


Figure 2.4. Failover to a redundant unit (based on [Hanmer 2013, Chapter 6.8])

Checkpoint

Save state periodically. Build in the capability to restore the system to the same state that was saved, without having to recreate the entire execution from startup to the point of the saved state [Hanmer 2013, Chapter 6.10].

Remote Storage

Store the saved checkpoints in a centrally accessible location. This enables a new processor to access the saved checkpoint which minimizes the period of unavailability [Hanmer 2013, Chapter 6.12].

2.6 Transport Protocols

The two most common transport protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is connection-based and needs a handshake to set up a connection. Furthermore, every message sent is acknowledged by the receiver. If an error occurs, the packages will be resent. Also, TCP preserves the order of the data packages. In contrast to TCP, UDP is a connectionless protocol. Therefore no handshake to establish the connection nor acknowledgments by the receiver is required. Erroneous packaged are ignored. TCP is more reliable, due to the package recovery. UDP is faster because no connection must be set up nor every package must be acknowledged.

2.7 The Actor Model

The Actor Model [Agha 1985] is a model for concurrent computation. The system is separated in concurrent units named actors and is based on the message passing communication pattern described in Section 2.4.1. Each actor can send and receive messages asynchronously to respectively from other actors. An actor stores received messages in a message inbox until it processes them following the FIFO-principle. Furthermore, an actor can change its behavior for the next messages it receives. Besides actors can create new actors.

2.8 The Akka Toolkit

Akka [Gupta 2012; Roestenburg et al. 2015] is a toolkit that implements the actor model. Actors are created inside an `ActorSystem`. The actor system is a container that manages the actors. It provides general configuration settings which apply to all actors of the system. To provide fault tolerance, an actor is the supervisor of its child actors. An actor can also become a supervisor of another actor by executing the `watch` method with the actor reference of the actor to be supervised as a parameter. Akka uses a *Let It Crash* fault tolerance model. Actors fail fast and are recovered by their supervisors. An actor system always has a root actor, which is the root supervisor of all actors created by the user.

The Akka toolkit contains the Akka Remote library. Akka Remote provides location transparency of actors, consequently a message can be sent to a remote actor by calling the `tell` method of its actor reference. Actor references are used the same for local and remote actors. Akka uses reliable buffered message passing. Furthermore Akka Remote supports the remote creation of actors on remote systems.

On top of Akka Remote, the Akka Cluster library provides cluster functionalities. At least one node must be defined as a seed node, which is an entry point to join the cluster. The seed nodes must be known to each actor system. In the cluster, an actor can subscribe to the cluster member events. If subscribed the actor is informed of events, such as actor systems joining or leaving the cluster. The cluster detects faults, e.g. a node become unreachable.

We use Akka for the implementation of the distributed Pipe-and-Filter architecture into TeeTime as we explain in Chapter 4.

2.9 Domain-specific Language

A domain-specific language (DSL) is a limited formal language specialized for a specific application domain [Fowler 2010]. A DSL provides a grammar to only solve problems in its domain, thus is simpler to learn than a general purpose language and helps to focus on the problem. Furthermore, a DSL can contribute to lower code redundancy by auto-generating boilerplate code and thus improve the code readability.

Chapter 3

Evaluation of Java Frameworks for Distributed Application Development

In this chapter, we evaluate the most suitable Java framework for distributed applications as the base for the distributed TeeTime implementation. For this purpose, we elaborated criteria the framework should comply to shown in Table 3.1. Each framework is reviewed based on these criteria.

3.1 Evaluation Criteria

We grouped the criteria into two categories, *Build Infrastructure* and *Features*. The *Build Infrastructure* category contains metadata about the framework and the *Features* category explicit functionalities.

3.1.1 Build Infrastructure Criteria

License represents the license under which the framework is provided. This license must be compatible with TeeTime's Apache 2 License so that the usage is faultless in legal terms.

Open Source frameworks provide the opportunity to extend and customize them if necessary. This may be the case when the framework lacks in some minor requirements, and a customization would be more suitable than another framework.

JDK Version defines the minimum required Java Development Kit (JDK) version to develop and run the application with the framework. The current major release is version 8. Currently, TeeTime supports Java down to version 6. This could be relevant if TeeTime should be used with legacy libraries with JDK dependencies which won't work with the current Java version. The usage of a framework with a higher JDK could restrict the usage of TeeTime. In fact, older Java versions are not supported anymore since April 2015 for version 7 and February 2013 for version 6[1]. Therefore this criterion only has an insignificant impact on the evaluation.

Latest activity indicates if the framework is still under maintenance, which is important concerning the possibility of bugs getting fixed. Moreover the implementation of necessary modifications due to possible changes in future JDK versions.

[1]http://www.oracle.com/technetwork/java/eol-135779.html

Table 3.1. Evaluation criteria

Criteria	Value Range	Requirements
Build Infrastructure		
License	*License name*	Apache 2.0 compatible
Open Source	yes/no	yes
Minimum JDK Version	*Version number*	Version 6
Latest activity	*Date*	newer than July 2016
Latest release	*Date (Version number)*	newer than July 2016
Features		
Communication Pattern	*Message Passing, RPC, DSM (opt. Protocol Name)*	Message Passing
Transport Protocol	TCP, UDP	TCP and UDP
Fault Tolerance	*Description of supported fault tolerance features*	Supervisor
Remote Deployment	yes/no	yes
Custom Serializer	yes/no	yes
Encryption	yes/no	yes

Latest release represents the most recent stable release version and release date of the framework. Newer beta versions received no consideration. The date shows the actuality of the development. Older dates perhaps indicate a longer period for bugs getting fixed.

3.1.2 Feature Criteria

Communication Patterns

Communication Patterns describes which of the three communication patterns *Message Passing*, *Remote Procedure Calls (RPC)* and *Distributed Shared Memory (DSM)* explained in Section 2.4 are covered by the framework. The usage of the distributed communication by TeeTime is limited to unidirectional data transmission between the nodes.

In this scenario, DSM has an unnecessary overhead. Instead of sending the data directly between the nodes it is accessed via its virtual address. The sender node has the data in memory, while the receiver node has not. Therefore the receiver node has to request the data from the sender which then gets replicated to the receiver's memory. Furthermore, the data occupies the local memory of the sender node although it is no longer needed.

RPC requests normally respond with the return type of the method call. The unidirectional data transmission results in `void` as a return type for all calls. Thus the abstractions made by RPC does not provide advantages towards message passing. Potentially the RPC abstractions decrease the optimization potential of the data transmission, concerning the Transport Protocol (Section 3.1.2) and the Custom Serializer (Section 3.1.2).

Therefore *Message Passing* is most suitable. This criterion supports Goal 1.1 (Section 1.2.1).

Transport Protocol

Transport Protocol defines which underlying protocol can be used for distributed communication. Possible protocols are TCP or UDP which is further explained in Section 2.6. The option to choose between TCP and UDP enables additional optimizations regarding reliability provided by TCP and speed enhancements provided by UDP.

This criterion supports Goal 1.1 (Section 1.2.1).

Fault Tolerance

Fault Tolerance contains the fault tolerance features the framework provide. Possible fault tolerance design patterns are described in Section 2.5.

This criterion supports Goal 1.2 (Section 1.2.1).

Remote Deployment

Remote Deployment indicates if the framework supports the automatic deployment of components to remote nodes. In this way, only a universal receiver must be deployed to the nodes without code related to the concrete Pipe-and-Filter architecture.

This criterion supports Goal 1.3 (Section 1.2.1).

Custom Serializer

Custom Serializer describes whether the usage of a custom serializer is possible. A custom serializer might be used to enhance the serializing speed depending on the performance of the existing serializer in the framework.

This criterion supports Goal 1.1 (Section 1.2.1).

Encryption

Encryption defines whether the framework supports the encryption of the data. Encryption is important because the nodes communicate via a possible insecure network, e.g. the Internet.

This criterion supports Goal 1.4 (Section 1.2.1).

3.2 Frameworks

In this section, we present 13 Java frameworks in alphabetical order which have relations to distributed communication or fault tolerance. In Section 3.3 a tabular overview of the frameworks is provided. The evaluation is at the level of December 10, 2016.

For comparison, the TeeTime framework is licensed under the Apache 2 license and is open source. The minimum JDK requirement is version 6. The latest activity was on November 22, 2016, and the latest non-snapshot version 2.1 was released on February 22, 2016.

3.2.1 Akka

The Akka framework is based on the Actor model explained in Section 2.7. Akka itself is described in Section 2.8. The project website is available at `http://akka.io`.

Build Infrastructure Criteria Akka is licensed under the Apache 2 license and is open source. The minimum JDK required is version 8. Akka is last edited on December 9, 2016, and the latest version 2.4.14 was released on November 22, 2016. Thus it is up-to-date.

Due to the same license as TeeTime, the usage of Akka in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to the newest JDK 8. Overall Akka fulfills the general criteria.

Feature criteria Akka's communication including the distributed part is based on message passing. The framework provides configuration options to either use TCP or UDP as the desired transport protocol. Moreover Akka provides several fault tolerance functionalities. Each actor is a supervisor of its child actors. As a supervisor, the actor can resume, restart or stop the child actors. Heartbeats are used to detect a failure. The detection is based on the Phi Accrual Failure Detector [Hayashibara et al. 2004]. Additionally, Akka provides message delivery reliability. Moreover, Akka supports remote deployment, custom serializer, and encryption of the communication. The remote deployment does not support remote class file loading, thus the class files must be available at the nodes. In conclusion Akka satisfies all feature criteria, with limitations regarding the remote deployment.

3.2.2 Apache Hadoop

Apache Hadoop provides distributed processing of large data sets in a cluster by MapReduce. The project consists of the modules Hadoop Distributed File System for data management, Hadoop YARN for job scheduling and cluster resource management and Hadoop MapReduce for data processing. The project website is available at `http://hadoop.apache.org`.

Build Infrastructure Criteria Hadoop is licensed under the Apache 2 license and is open source. The minimum JDK required is version 7. Hadoop is last edited on December 10, 2016, and the latest version 2.7.3 was released on August 25, 2016. Thus it is in the required date range.

Due to the same license as TeeTime, the usage of Hadoop in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to JDK 7 and above. Overall Hadoop fulfills the general criteria.

Feature criteria Apache Hadoop's distributed communication is based on RPC and the distributed shared memory HDFS (Hadoop Distributed File System). TCP is used, and there is no option to change the transport protocol to UDP. Hadoop provides a Health Monitor which uses Heartbeats to detect failures. Failed tasks can be reexecuted due to the master/worker architecture. The master is a single point of failure and does not provide fault tolerance. Hadoop neither provides remote deployment, nor custom serializer. Encryption is provided by HDFS.

With MapReduce Hadoop provides its own data processing architecture, thus is not suitable for TeeTime as a base. Additionally, the communication via the HDFS adds unnecessary overhead to the desired distributed TeeTime architecture. HDFS always write and reads via the HDFS instead of directly sending messages between the nodes. Overall Hadoop does not meet the feature criteria.

3.2.3 Apache River

Apache River is an implementation of a service orientated architecture which communicates via RPC. River helps to construct distributed systems consisting of services. The project website is available at `https://river.apache.org`.

Build Infrastructure Criteria River is licensed under the Apache 2 license and is open source. The minimum JDK required is version 6. River is last edited on December 10, 2016, and the latest version 2.2.3 was released on February 21, 2016, thus is not inside the required date range. River has a huge release cycle (the previous version was released in 2013).

Due to the same license as TeeTime, the usage of River in TeeTime is permitted under copyright law. The minimum JDK version 6 add no restrictions to TeeTime. Overall River does not meet the general criteria due to the latest release date and the general release cycle.

Feature criteria River provides the communication protocol JERI which is based on RPC. TCP is used, and there is no option to change the transport protocol to UDP. Fault tolerance functionalities are not available. River provides remote deployment, but no custom serializer.

The communication can be encrypted via plain-SSL or HTTPS. Overall River does not meet the feature criteria, especially the lack of fault tolerance.

3.2.4 Apache Spark

Apache Spark is a cluster computing framework. It is based on the resilient distributed dataset (RDD) data structure, which is a read-only multiset of data items. Spark is an improvement over MapReduce to support in-memory processing with RDD instead of using HDFS for data sharing. Spark includes a Spark Streaming component which processes mini-batches of data. The project website is available at `http://spark.apache.org`.

Build Infrastructure Criteria Spark is licensed under the Apache 2 license and is open source. The minimum JDK required is version 7. Spark is last edited on December 10, 2016, and the latest version 2.0.2 was released on November 14, 2016. Thus it is up-to-date.

Due to the same license as TeeTime, the usage of Spark in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to JDK 7 and above. Overall Spark fulfills the general criteria.

Feature criteria Spark uses Remote Procedure Calls for distributed communication. There is no information about the transport protocol. To provide fault tolerance, the remote storage (RDD) prevents data loss through data replication. Checkpoints are used, and data is stored to provide zero-data-loss. Data record processing can be guaranteed. Additionally, nodes can be recovered from failures. Spark supports manual remote deployment via the CLI tool spark-submit, a custom serializer, and encryption.

Same as MapReduce Spark provides its own data processing architecture, therefore does not fit in well with TeeTime. Although most of the criteria are fulfilled, a programmatic remote deployment is desired and not one via a CLI tool. Thus Spark does not meet the general criteria.

3.2.5 Apache Storm

Apache Storm is a real-time stream processing framework. The project website is available at `http://storm.apache.org`.

Build Infrastructure Criteria Storm is licensed under the Apache 2 license and is open source. The minimum JDK required is version 7. Storm is last edited on December 8, 2016, and the latest version 1.0.2 was released on August 10, 2016, thus it is in the required date range.

Due to the same license as TeeTime, the usage of Storm in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to JDK 7 and above. Overall Storm fulfills the general criteria.

Feature criteria Storm uses an own protocol for communication, named Thrift. Thrift provides both RPC and message passing. There is no information about the transport protocol available. Storm provides a supervisor, which uses heartbeats to detect node failures. The worker can be restarted, or the task can be reassigned to other nodes. Message processing is guaranteed. Storm provides remote deployment, but neither the option to use a custom serializer nor encryption.

Same as MapReduce and Spark Storm already has its own data processing architecture; thus an integration of TeeTime does not seem advisable. Hence Storm does not meet the feature criteria.

3.2.6 Apache ZooKeeper

Apache ZooKeeper is a coordination server which provides features for configuration, synchronization, leader election and notification in distributed systems. The project website is available at `https://zookeeper.apache.org`.

Build Infrastructure Criteria ZooKeeper is licensed under the Apache 2 license and is open source. The minimum JDK required is version 7. ZooKeeper is last edited on December 3, 2016, and the latest version 3.4.9 was released on September 3, 2016, thus is in the date range. Due to the same license as TeeTime, the usage of ZooKeeper in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to JDK 7 and above. Overall ZooKeeper fulfills the general criteria.

Feature criteria ZooKeeper as a coordination server does not provide one of the desired features but supports the previously presented Apache frameworks.

ZooKeeper can only be used together with one of these Apache frameworks. The feature criteria are thus not reviewed in detail. ZooKeeper meets the general criteria and therefore is possible to use with the other frameworks.

3.2.7 Atomix (+ Copycat + Catalyst)

Atomix is a software stack for distributed systems. It consists of Atomix as a coordination framework, Copycat as a state machine replication framework and Catalyst as an I/O & binary serialization framework. The project website is available at `http://atomix.io`.

Build Infrastructure Criteria Atomix, Copycat, and Catalyst are licensed under the Apache 2 license and are open source. The minimum JDK required is version 8. Atomix is last edited on November 3, 2016. The latest version 1.0.0 RC9 was released on June 23, 2016. Thus it is not in the required date range. Since the release, only one bug has been fixed.

Due to the same license as TeeTime, the usage of Atomix in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to the newest JDK 8.

Since Atomix latest release is not in the required date range, it does not meet the general criteria.

Feature criteria Atomix uses message passing for distributed communication. TCP is used as the transport protocol without the support for UDP. Atomix fault tolerance is based on the architectural pattern *someone in charge*. Atomix uses a leader election algorithm which will be performed when no leader is available, in the case of a leader failure or a network partition. Heartbeats are used to detect node failures. Passive nodes which already contains state copies of other nodes or reserve nodes can be provided for failover. Atomix does not support remote deployment. The option to use a custom serializer and to enable encryption is available.

The general architecture Atomix provide would suit well with TeeTime, but due to the lack of UDP support and remote deployment, Atomix does not meet the feature criteria.

3.2.8 Axon Framework

Axon Framework is based on the Command Query Responsibility Segregation (CQRS) architectural pattern which separates commands to update information and queries to read information. Furthermore, Axon builds upon JGroups (Section 3.2.10). The project website is available at `http://www.axonframework.org`.

Build Infrastructure Criteria Axon is licensed under the Apache 2 license and is open source. The minimum JDK required is version 6. Axon is last edited on December 9, 2016, and the latest version 2.4.5 was released on July 25, 2016, thus is in the date range.

Due to the same license as TeeTime, the usage of Axon in TeeTime is permitted under copyright law. The minimum JDK version 6 add no restrictions to TeeTime. Overall Axon fulfills the general criteria.

Feature criteria Axon is based on JGroups, therefore uses the same message passing for distributed communication. Both TCP and UDP can be configured via the JGroups configuration. There are no fault tolerance features mentioned. We assume that only the fault tolerance features of JGroups are provided. Remote deployments are not available. A custom serializer can be used. Encryption is not mentioned but may be configured via the JGroups configuration.

Overall Axon does not meet the feature criteria.

3.2.9 Hystrix

Hystrix is a fault tolerance library. The project website is available at `https://github.com/Netflix/Hystrix`.

Build Infrastructure Criteria Hystrix is licensed under the Apache 2 license and is open source. The minimum JDK required is version 8. Hystrix is last edited on December 6, 2016, and the latest version 1.5.8 was released on November 10, 2016, thus is up-to-date.

Due to the same license as TeeTime, the usage of Hystrix in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to the newest JDK 8. Overall Hystrix fulfills the general criteria.

Feature criteria Hystrix is specialized on fault tolerance. Thus none of the other criteria are fulfilled. Hystrix provides mechanisms to prevent servers from overload. It follows the fail fast and rapid recovery paradigm. If possible, it provides fallback to a previous state and graceful degradation, which is the ability to maintain limited functionality in case of a failure.

Since only fault tolerance is provided, Hystrix does not meet the feature criteria. It may be used to extend other frameworks.

3.2.10 JGroups

JGroups is a framework for reliable group communication and membership management. The project website is available at `http://jgroups.org`.

Build Infrastructure Criteria JGroups is licensed under the Apache 2 license and is open source. The minimum JDK required is version 7. JGroups is last edited on December 7, 2016, and the latest version 3.6.11 was released on September 19, 2016. Thus it is in the required date range.

Due to the same license as TeeTime, the usage of JGroups in TeeTime is permitted under copyright law. The minimum JDK version restricts the development to JDK 7 and above. Overall JGroups fulfills the general criteria.

Feature criteria JGroups utilize message passing for distributed communication. TCP and UDP as transport protocols are configurable. Fault detection is implemented with detection of crashed nodes via heartbeats, removal of crashed nodes from the cluster and removal notifications for the other nodes. A supervisor can be used by implementing own rules. Neither remote deployment nor a custom serializer option is provided. Encryption can be configured.

Overall JGroups does not meet the feature criteria.

3.2.11 JPPF

JPPF is the abbreviation for Java Parallel Processing Framework. JPPF is based on a master/worker architecture. The project website is available at `http://www.jppf.org`.

Build Infrastructure Criteria JPPF is licensed under the Apache 2 license and is open source. The minimum JDK required is version 7. JPPF is last edited on December 7, 2016, and the latest version 5.2.3 was released on November 27, 2016, thus is up-to-date.

Due to the same license as TeeTime, the usage of JPPF in TeeTime is permitted under copyright law.The minimum JDK version restricts the development to JDK 7 and above. Overall JPPF fulfills the general criteria.

Feature criteria JPPF's distributed communication is based on message passing. Only TCP can be used as the transport protocol. JPPF provides failover by recovering submitted jobs. A persistent store is used as a remote storage to save state information. Remote deployment, the option for using a custom serializer and the option to use encryption are available.

JPPF is missing the support of UDP as a transport protocol option. JPPF uses a master/-worker architecture by distributing tasks to nodes. These nodes do not communicate with each other. Therefore this architecture is not suitable for TeeTime. JPPF provide P2P of the master nodes. Nonetheless, the worker nodes only communicate with the master. Thus no full P2P architecture can be provided.

3.2.12 Orbit

Orbit is based on JGroups (Section 3.2.10) and implements the Actor model (see in Section 2.7) on top of it. The project website is available at `https://github.com/orbit/orbit/wiki`.

Build Infrastructure Criteria Orbit is licensed under the BSD 3-Clause license and is open source. The minimum JDK required is version 8. Orbit is last edited on November 14, 2016, and the latest version 0.9.12 was released on November 14, 2016, thus is up-to-date.

The license differs from TeeTime. The BSD 3-Clause is compatible with Apache 2. The minimum JDK version restricts the development to the newest JDK 8. Overall Orbit fulfills the general criteria.

Feature criteria Orbit implements the actor model, and the communication is based on JGroups. Therefore it also provides the option to use TCP or UDP. Fault tolerance functionalities are not mentioned. Same as JGroups neither remote deployment nor a custom serializer is supported. Encryption is not mentioned but may be used via the underlying JGroups configuration.

Overall Orbit does not meet the feature criteria.

3.2.13 Quasar

Quasar is based on JGroups and implements the Actor model (see in Section 2.7) on top of it. The project website is available at `http://www.paralleluniverse.co/quasar`.

Build Infrastructure Criteria Quasar is licensed both under the Eclipse Public License v1.0 and the GPL 3.0 license and is open source. The minimum JDK required is version 7. Quasar is last edited on December 10, 2016, and the latest version 0.7.7 was released on December 2, 2016, thus is up-to-date.

The license differs from TeeTime. The GPL 3.0 license is incompatible with Apache 2 and prohibits the use in Apache 2 software. The alternative Eclipse Public License v1.0 could be utilized when labeled appropriately. The minimum JDK version restricts the development to JDK 7 and above. Overall Quasar fulfills the general criteria. In the case Quasar is used we must pay attention to the referencing of the Eclipse Public License.

Feature criteria Similar to Orbit Quasar implements the actor model as well. Quasars distributed communication is also based on JGroups message passing. TCP and UDP are configurable. No fault tolerance features are described. Same as JGroups neither remote deployment nor a custom serializer is supported. Encryption is not mentioned but may be used via the underlying JGroups configuration.

Overall Quasar does not meet the feature criteria.

3.3 Framework Overview and Conclusion

From all 13 examined frameworks only Akka fulfills every criterion. Since no other framework fulfills the criteria, no more profound comparison is necessary. Thus, we decide to use Akka as a basis for further developments.

Some features the frameworks provide might be missing. These features may not be documented or could not be detected during the research. Consequently, the presented information are to be understood as a minimum quantity of features the frameworks provide.

Table 3.2. Evaluation Overview of Java Frameworks: General Criteria

Framework	**URL**	**License**	**Open Source**	**JDK**	**Latest activity**	**Latest Release**[a]
Akka	`http://akka.io`	Apache 2	yes	8	09.12.2016	22.11.2016 (2.4.14)
Apache Hadoop	`http://hadoop.apache.org`	Apache 2	yes	7	10.12.2016	25.08.2016 (2.7.3)
Apache River	`https://river.apache.org`	Apache 2	yes	6	10.12.2016	21.02.2016 (2.2.3)
Apache Spark	`http://spark.apache.org`	Apache 2	yes	7	10.12.2016	14.11.2016 (2.0.2)
Apache Storm	`http://storm.apache.org`	Apache 2	yes	7	08.12.2016	10.08.2016 (1.0.2)
Apache ZooKeeper	`https://zookeeper.apache.org`	Apache 2	yes	7	03.12.2016	03.09.2016 (3.4.9)
Atomix (+ Copycat + Catalyst)	`http://atomix.io`	Apache 2	yes	8	03.11.2016	23.06.2016 (1.0.0 RC9)
Axon Framework	`http://www.axonframework.org`	Apache 2	yes	6	09.12.2016	25.07.2016 (2.4.5)
Hystrix	`https://github.com/Netflix/Hystrix`	Apache 2	yes	8	06.12.2016	10.11.2016 (1.5.8)
JGroups	`http://jgroups.org`	Apache 2	yes	7	07.12.2016	19.09.2016 (3.6.11)
JPPF	`http://www.jppf.org`	Apache 2	yes	7	07.12.2016	27.11.2016 (5.2.3)
Orbit	`https://github.com/orbit/orbit/wiki`	BSD 3-Clause License	yes	8	14.11.2016	14.11.2016 (0.9.12)
Quasar	`http://www.paralleluniverse.co/quasar`	Eclipse Public License v1.0 / GPL 3.0	yes	7	10.12.2016	02.12.2016 (0.7.7)

[a]last check December 10, 2016

Table 3.3. Evaluation Overview of Java Frameworks: Feature Criteria (Part 1/2)

Framework	**Distributed Communication**	**Transport Protocol**	**Fault Tolerance**
Akka	Message passing	TCP, UDP	Supervisor, Heartbeats with Phi Accural Failure Detector, Message Delivery Reliability
Apache Hadoop	RPC, DSM (HDFS)	TCP	Health Monitor with Heartbeats, Task reexecution
Apache River	RPC (JERI)	TCP	-
Apache Spark	RPC	*No info.*	data replication, checkpoints, data processing guarantee, node recovery
Apache Storm	Message passing, RPC (Thrift)	*No info.*	supervisor, heartbeats, worker restart and reassignment, message processing guarantee
Apache ZooKeeper	*Not implemented*	*No info.*	-
Atomix (+ Copycat + Catalyst)	Message passing	TCP	leader election, heartbeats, failover (passive and reserve nodes)
Axon Framework	Message passing	TCP, UDP	-
Hystrix	*Not implemented*	*No info.*	fault detection, rapid recovery, fallback, graceful degradation
JGroups	Message passing	TCP, UDP	Hearbeats, removal of crashed nodes, removal notifications, supervisor (with custom rules)
JPPF	Message passing	TCP	failover, recovery, persistent store (remote storage for recovery)
Orbit	Message passing	TCP, UDP	-
Quasar	Message passing	TCP, UDP	-

Table 3.4. Evaluation Overview of Java Frameworks: Feature Criteria (Part 2/2)

Framework	**Remote Deployment**	**Custom Serializer**	**Encryption**
Akka	yes (no remote class file loading)	yes	yes
Apache Hadoop	yes	no	yes
Apache River	yes	no	yes
Apache Spark	yes (via CLI tool)	yes	yes
Apache Storm	yes	no	no
Apache ZooKeeper	no	no	no
Atomix (+ Copycat + Catalyst)	no	yes	yes
Axon Framework	no	yes	no[a]
Hystrix	no	no	no
JGroups	no	no	yes
JPPF	yes	yes	yes
Orbit	no	no	no[a]
Quasar	no	no	no[a]

[a]maybe via underlying JGroups configuration

Chapter 4

Implementation of a Distributed Pipe-and-Filter Architecture

Based on the evaluation results in Chapter 3 the distributed Pipe-and-Filter architecture is implemented by integrating the Akka framework into TeeTime.

An overview of the architecture is shown in Figure 4.1. TeeTime components are highlighted in teal and Akka components are highlighted in yellow. The architecture is divided into a master client and multiple worker nodes. A master is a control unit whereas the workers execute the TeeTime pipeline.

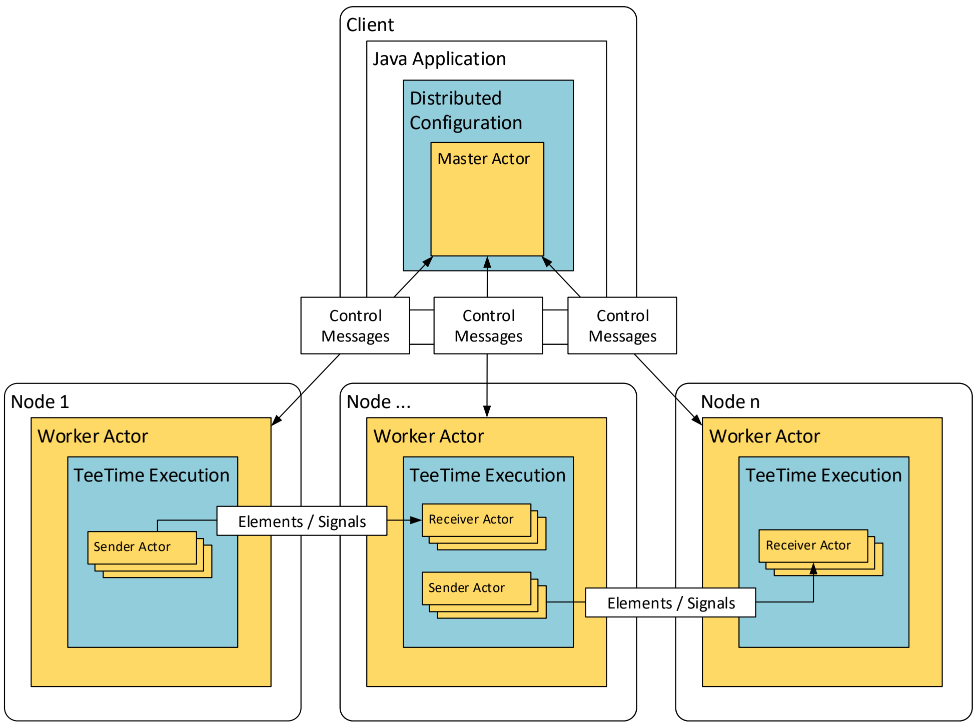


Figure 4.1. Distributed Pipe-and-Filter Architecture Overview

We describe the communication between the workers in Section 4.1. Afterwards we explain the distributed configuration in Section 4.2. Then we elucidate the remote deployment and execution of the worker nodes in Section 4.3. Finally, we outline the fault tolerance of the system in Section 4.4

4.1 Implementation of Distributed Communication

When splitting up a pipeline and deploying it to several nodes, some stages need to communicate via the network. We can implement the network communication in various locations in TeeTime, either in

▻ the stages (Figure 4.2a),

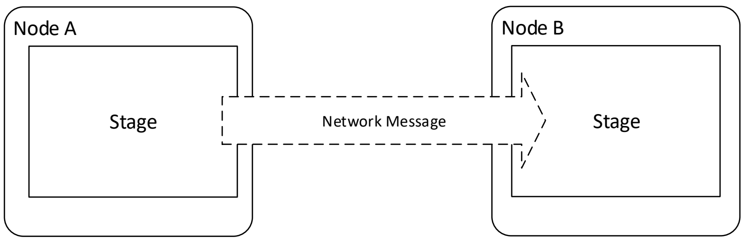
▻ the ports (Figure 4.2b), or

▻ the pipes (Figure 4.2c).

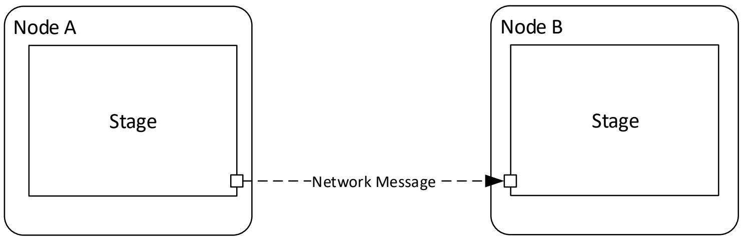
We avoid the direct communication through stages because the purpose of stages in a Pipe-and-Filter architecture is data processing and not transportation. When a stage should be used both in a non-distributed and distributed context the implementation of communication inside the ports would be an obstacle. It would be necessary to override the ports in one of these cases with either the distributed or non-distributed variant. To provide universal usage of stages the distributed communication is implemented in a pipe, named `DistributedPipe`, and is consequently stage independently.

4.1.1 Distributed Pipe

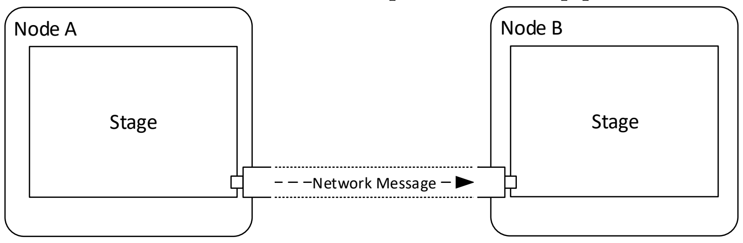
The `DistributedPipe` is based on the `AbstractSynchedPipe` in TeeTime and therefore provides two queues: one to store elements and one to store TeeTime signals. Instead of a normal TeeTime configuration, a distributed configuration is used, which is explained in more detail in Section 4.2. The `DistributedPipe` is only utilized to connect stages between different nodes. Each worker runs a separate Java VM and is in most cases separated via the network. Therefore a pipe can not consist of a single pipe instance. Instead, there is a sender and receiver stub which together form the pipe. Each node instantiates the `DistributedPipe`, thus it is once used as the sender and once as the receiver stub. Therefore a sender actor is integrated into the sender stub. The sender actor gets notified, when a new element or signal is added to the queue of the pipe, pulls it from the belonging queue and sends it to the corresponding receiver actor. This receiver actor is integrated to the receiver stub. When an element or signal is received by the actor, it is added to the queue and is available for the stage this pipe is connected to. The addition of the actors is performed during runtime and is shown in Figure 4.3.



(a) Communication via stages: sender stage without output port and receiver stage without input port



(b) Communication via ports without a pipe



(c) Communication via a pipe

Figure 4.2. Possible scenarios of distributed communication

4.1.2 Auto-Discovery of the Receiver

During configuration, an identifier for the receiver actor is automatically specified, which is used by the sender actor to auto-discover the receiver in the cluster. Automatic discovery of the target actor highly improves the reusability. In a previous approach, we hard-coded the node IP addresses and ports into the configuration, which prevented the reuse of the configuration on a divergent node cluster with different IP addresses. The auto-discovery is shown in Figure 4.4.

Akka provides a Publish-Subscribe mediator. The mediator is accessible in the whole cluster via its identifier without knowing the mediator’s exact location. Each receiver actor registers to the mediator with the specified identifier of the actor. Meanwhile, the corresponding sender actor requests the actor path via the mediator. The sender actor repeats the request periodically until it receives an actor path. The path depends on the

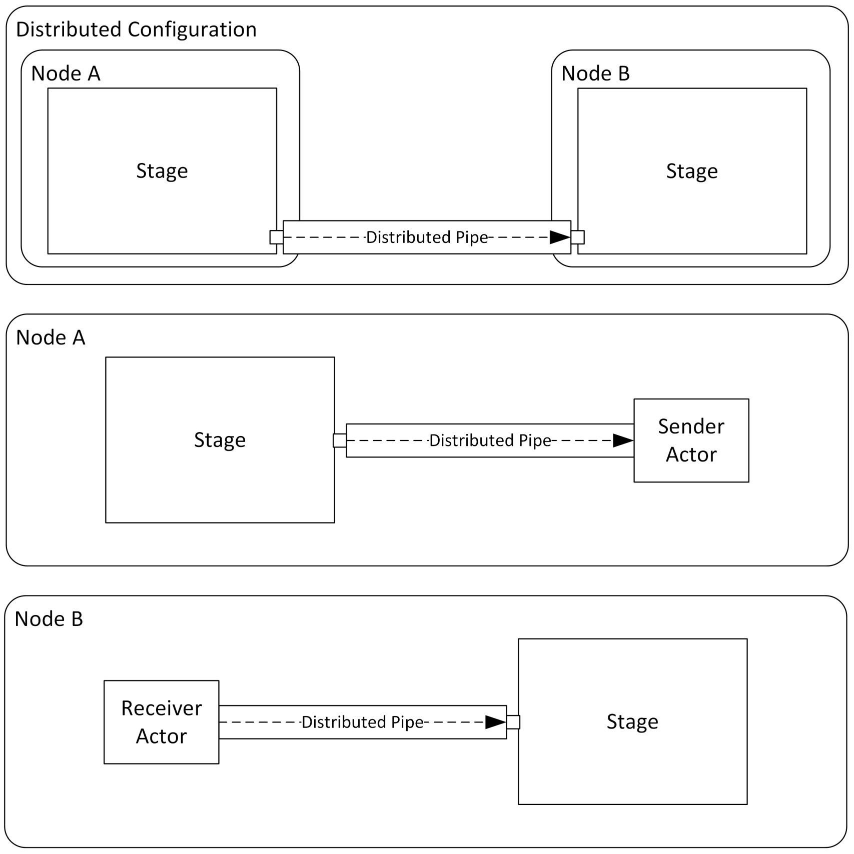


Figure 4.3. Adding sender and receiver actors to the distributed pipe

transport protocol specified for the pipe in the configuration, which is either TCP, UDP or SSL over TCP. Encryption over UDP is not available because SSL builds upon TCP. The actor reference is resolved via this path. This actor reference can now be used to communicate directly with the receiver without using the bypass over the mediator.

We perform the auto discovery before the TeeTime execution starts. In a previous approach, the TeeTime execution started after all actors were created. At this moment the actor reference of the receiver was not resolved yet. When the preceding TeeTime stages processed a high amount of elements before the actor path was received the actor receives many notification messages for new signals and elements to be pulled. Therefore the actor tries to send the elements, which is not possible until a receiver actor reference is resolved. The `ActorPathMessage` is not processed until all previous notification messages are processed. The worst case is that the first worker processes all elements and stores them in the queue of the `DistributedPipe` before the receiver reference is resolved and the

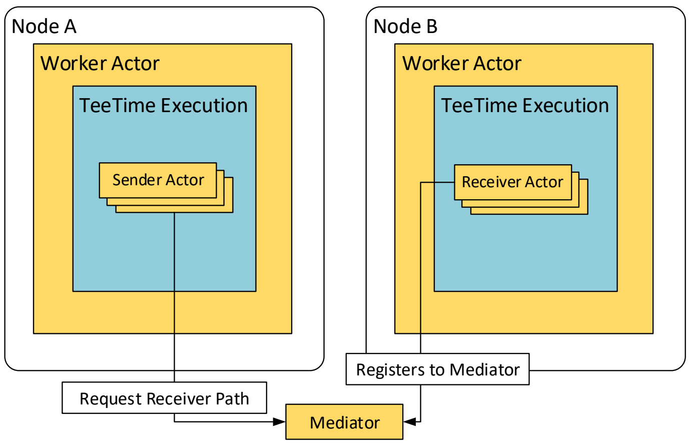


Figure 4.4. Auto-Discovery of the receiver

actor starts to transmit the elements. Tests showed that this leads to significantly higher execution times and the queue consumes substantial more memory.

4.1.3 Handling the TeeTime Terminating Signal

When a producer stage in TeeTime finishes, it sends a terminating signal to the following stages and sets its status to terminated. As soon as all stages of a node are set to terminated, the `Execution` instance of the node terminates. It might be the case that the sender actor still sends elements after the `Execution` terminated. Consequently, it would be too early to terminate the whole worker directly after the `Execution` terminated. To guarantee that all messages are sent to the receiver, the sender actor waits until the receiver actor processed the terminating signal, after which the receiver answers with a `ShutdownWorker` signal. No further elements will be sent to the receiving actor. It might be more beneficial to free the resources allocated by this actor and provide them to the remaining stages of the node. Therefore the receiver actor also terminates. After all actors of a node have terminated the whole worker terminates. The worker notifies the master about its shutdown so that the master does not handle the shutdown as a worker failure. If all workers have terminated, the master sets its own status to terminated.

4.1.4 Message Serialization

When an actor sends a Java object to a remote actor via the network, the object must be serialized by the sender and deserialized by the receiver. By default, Akka uses the standard Java serialization. The Java serialization requires every object which should be serialized to implement the `java.io.Serializable` interface. Otherwise, it denies the serialization. The constraint to implement the `Serializable` interface may be problematic when objects rely on third party libraries which do not use the interface and couldn't be modified, because they are only available as binary class files. This constraint would prevent the conversion from a non-distributed to a distributed Pipe-and-Filter architecture when using objects without the `Serializable` interface.

We use the Kryo serializer[1] to provide serialization for all objects. The official Akka documentation recommends the Kryo serializer and discourages the use of the standard Java serializer. Kryo is integrated via the Chill Akka extension[2] provided by Twitter as shown in Line 2 of Listing 4.1. Instead of implementing the `Serializable` interface Kryo requires a serialization binding for the root object. Therefore we implemented a `TeeTimeDataMessage` and a `TeeTimeSignalMessage` which wraps the actual elements and signals and are bounded to Kryo as shown in Line 5 and 6 of Listing 4.1. The wrapper classes improve the usability so that the users do not have to define bindings for all objects they want to send to a remote actor. Kryo may also provide performance benefits compared to the standard Java serializer. The evaluation of the serializer performance is not in the scope of this thesis.

```
1 serializers {
2    kryo = "com.twitter.chill.akka.AkkaSerializer"
3 }
4 serialization-bindings {
5    "teetime.framework.distributed.message.TeeTimeDataMessage" = kryo
6    "teetime.framework.distributed.message.TeeTimeSignalMessage" = kryo
7    ...
8 }
```

Listing 4.1. Akka Configuration for Serialization

4.2 Implementation of a Single Configuration for a Distributed Execution

The entire distributed Pipe-and-Filter architecture is defined in a single distributed configuration. Each worker executes its own TeeTime configuration which is automatically generated from the distributed configuration. The single distributed configuration enhances the transparency compared to defining each of this configurations independently. A single configuration provides a full overview of the whole architecture and in particular the connections between the nodes. Defining a distributed configuration is very similar to a non-distributed configuration as shown in Listing 4.2. The differences are highlighted in different colors. Instead of extending the `Configuration` class, we extend

[1] https://github.com/EsotericSoftware/kryo
[2] https://github.com/twitter/chill

the AbstractDistributedConfiguration like shown in Line 1 highlighted in red. The stage declarations are unchanged. An important addition is createNodeForStages shown in Line 9, 10 and 11 highlighted in teal. With this method a configuration is created for a worker node and the stages are assigned. A stage can only be added to one node, otherwise a IllegalArgumentException is thrown. As we mentioned in Section 4.1.2 the configuration does not specify the IP address and ports of a node to improve reusability. In case a node configuration relates to a specific server to provide for example data locality an optional node identifier can be used to express the relation between a configuration and a worker node. For instance the identifier Node1 is specified in Line 9 highlighted in purple for the configuration. This configuration is only deployable to a worker which also has the identifier Node1. We further explain the deployment in Section 4.3. Furthermore we extend connectPorts with an optional TransportProtocol parameter to specify the transport protocol for this connection. For example the TCP protocol is defined in Line 14. Additional option values are TransportProtocol.UDP and TransportProtocol.SSL for the UDP and, respectively, the SSL protocol. The source and target stage must be assigned to different nodes to allow the TransportProtocol parameter. Otherwise a IllegalStateException occurs. The TCP protocol is chosen by default when the TransportProtocol parameter is left out for a connection between different nodes as shown in Line 15.

```
 1 public class DistributedConfiguration extends AbstractDistributedConfiguration  {
 2    public DistributedConfiguration(){
 3       RandomStringGeneratorStage stage1 = new RandomStringGeneratorStage(...);
 4       CPULoadGeneratorStage stage2 = new CPULoadGeneratorStage(...);
 5       CPULoadGeneratorStage stage3 = new CPULoadGeneratorStage(...);
 6       CPULoadGeneratorStage stage4 = new CPULoadGeneratorStage(...);
 7       EndStage stage5 = new EndStage();
 8
 9       createNodeForStages("Node1", stage1, stage2);
10       createNodeForStages(stage3);
11       createNodeForStages(stage4, stage5);
12
13       connectPorts(stage1.getOutputPort(), stage2.getInputPort());
14       connectPorts(stage2.getOutputPort(), stage3.getInputPort(),
             TransportProtocol.TCP);
15       connectPorts(stage3.getOutputPort(), stage4.getInputPort());
16       connectPorts(stage4.getOutputPort(), stage5.getInputPort());
17    }
18 }
```

Listing 4.2. Distributed Configuration Example

To transform a non-distributed configuration to a distributed one it is sufficient to change the superclass from Configuration to AbstractDistributedConfiguration and assign all stages to nodes via `createNodeForStages`.

Besides the procedure of executing the configuration changes. To run a `Configuration` in a non-distributed scenario an `Execution` is used. This `Execution` can either be instantiated and executed embedded in a Java program or executed as a standalone Java application. To execute TeeTime in a distributed setting an `AbstractDistributedConfiguration` is created embedded in a Java program instead of a normal `Configuration` as shown in Listing 4.3. Currently it is not yet possible to execute the `AbstractDistributedConfiguration` as a standalone application.

```
1 AbstractDistributedConfiguration config =
      AbstractDistributedConfiguration.getConfig(configName, configArgs);
2 config.executeDistributed(host, port);
3 config.isTerminated();
```

Listing 4.3. Execution of a Distributed Configuration

The `AbstractDistributedConfiguration` provides a `getConfig` method to get a configuration instance by name shown in Line 1. Alternatively a distributed configuration can be instantiated directly via its constructor e.g. a `new ExampleDistributedConfig()`. To execute the configuration `executeDistributed()` is called in Line 2. The host IP and port are passed to create the master actor system listening on this host and port. The execution is non-blocking. To check whether the execution is terminated the method `isTerminated()` can be called on the configuration as shown in Line 3. Workers process the actual TeeTime pipeline. These workers need to be executed as standalone applications on the nodes.

4.3 Implementation of Remote Deployment and Remote Execution

The distributed Pipe-and-Filter architecture is divided between a master and some workers. The master must be embedded in a Java program. For the workers, we provide a `RemoteSystem` as part of TeeTime, which needs to be executed as a standalone Java application on the nodes (the start command of the remote system is shown in Listing B.1). The remote system must be deployed to each node manually. Akka does not support distributed class loading. Therefore the remote system requires access to the class files of the application on each node as well so that the worker actor can instantiate the configuration and the stages assigned to the node. Each worker runs a worker actor who handles the control messages of the master. The deployment is shown in Figure 4.5. The single steps are marked with numbers in a red circle. The worker actor automatically registers the worker node to the master either as a generic node or a specific node if a node identifier is specified (Step 1). The master then assigns a configuration to the worker depending on the identifier or in the case of a generic node successively a configuration of the unspecified ones (Step 2). Thus the user has the choice between exactly specifying which configuration should be

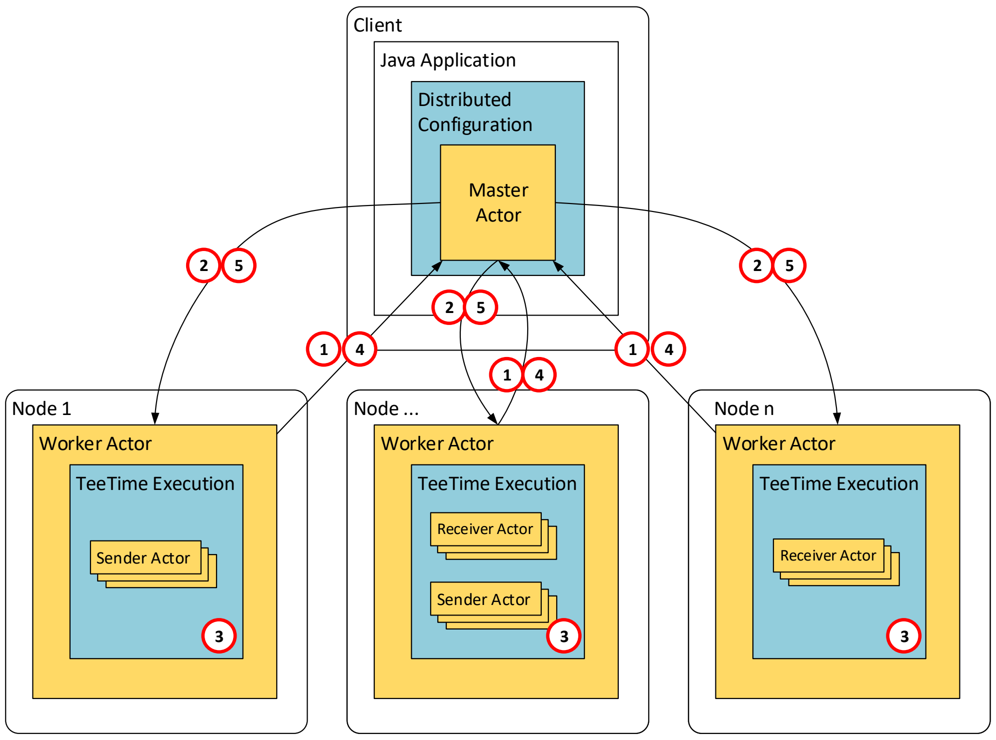


Figure 4.5. Remote deployment and remote execution

deployed on which worker or letting TeeTime automatically deploy configurations to free workers. The workers only receive the configuration name and retrieve the configuration from the local distributed configuration instance. Based on the assigned configuration sender and receiver actors are created, and a TeeTime `Execution` is instantiated (Step 3). When all sender actors received their corresponding receiver actor references as described in Section 4.1.2 the worker actor notifies the master that the node is ready (Step 4). As soon as all configurations are assigned to a worker node and all worker acknowledged that they are ready a signal is sent to start the execution (Step 5).

4.4 Implementation of Fault Tolerance

To demonstrate the importance of providing fault tolerance we consider an example distributed Pipe-and-Filter architecture without any explicit fault tolerance. In the case of a node failure preceding nodes further on try to send elements to the crashed node. TCP as a transport protocol guarantees message delivery by requiring an acknowledgment (explained in Section 2.5.2) for every message. If no acknowledgment is received, the message is resent. When TCP is used the sender actor indefinitely retries to send the elements. Thus no elements are discarded, and the queue fills up. The maximum queue size is unlimited thus the queue may exceed the system memory when many elements are processed. High memory consumption could affect the availability of the entire node and furthermore influences other services provided by the node. Succeeding nodes remain running without receiving new elements and consequently waste system resources. Human interaction is required to recognize the failure and stop the system manually.

To prevent the described behavior, we provide fault tolerance in our distributed Pipe-and-Filter architecture. The applied fault tolerance patterns are described in Section 2.5. We minimize human interaction by putting the master node partially in charge of handling failures in the system. The master is a fault observer and observes all worker nodes by providing a system monitor which periodically sends heartbeats to the worker. To offer realistic thresholds for the heartbeat timeouts, Akka uses a Phi Accrual Failure Detector [Hayashibara et al. 2004]. This failure detector uses the historical arrival times of heartbeat responses and examines its deviation instead of predefining a fixed timeout. When the worker is still capable of sending messages after an exception occurred, it will forward the exception message to the master. If a worker crashes, the master recognizes that the worker becomes temporarily unreachable. After some time it will declare the worker as permanently unavailable and administer the termination of the entire architecture. As mentioned, the master is not fully in charge of the whole system. Each actor implements a `SupervisorStrategy` as an error handler. Exceptions are forwarded to the master, but afterward, the node terminates itself instead of waiting for a control message from the master. In case the master crashes all worker nodes decide for itself that the master is permanently unavailable and terminate themselves. The different fault handling procedures are shown and evaluated in Section 6.4. Instead of putting only the master in charge of the fault handling it is distributed to all nodes to guarantee a graceful termination of the whole architecture. This is done because termination is the most reasonable conclusion.

In fact, the desired fault tolerance behavior would be a recovery from the failed state and resume the execution. The recovery could be either done by completely restart the whole system or by rollback, roll-forward or return to a reference point. Except for the restart, the state of the whole system must be recovered to a consistent state in these cases. The state can be periodically saved at checkpoints to a remote storage. The periodic saving is problematic in a Pipe-and-Filter architecture because the system has to persist the state of each stage and the objects in the pipes all in the same moment. To guarantee that while executing the stages concurrently the whole architecture must halt, store its state and continue the execution. In a distributed scenario the halt must be synchronized between the nodes before the state can be stored. The synchronization would lead to a massive execution overhead. Thus only the complete system restart may be useful. Depending on the error the failure may occur again caused by the same invalid data input. Therefore we decide to shutdown the system in case of a failure and let the user choose if he wants to restart the execution after checking the log files.

Chapter 5

Extending the TeeTime Domain-specific Language

We implement the distributed communication through pipes as described in Section 4.1, therefore the stages will be unaffected, and thus only the DSL for the configuration must be extended. We implement the TeeTime DSL as an Eclipse plug-in based on the Xtext framework[1].

Zloch [Zloch 2016] presents a TeeTime Configuration DSL, which Tavares de Sousa developed further. Listing 5.1 shows a simple example configuration based on the current level of development. Line 1 and 2 defines the import statements. Afterward, Line 3 specifies the configuration constructor. In this example, no constructor parameters are passed. Inside the constructor, Line 5 and 6 define the stage declarations and initializations. The `active` statement declares the `CollectorSink` as active, which means it will be executed in its own thread. Line 8 specifies the connection between the stages. The example DSL file is located in the `src` folder in the package `teetime.test`. The generator adds the package to the Java class depending on the package location of the DSL file.

```
1 import teetime.stage.InitialElementProducer
2 import teetime.stage.CollectorSink
3 Config(){
4    // stages
5    InitialElementProducer producer()
6    active CollectorSink consumer()
7    // connections
8    producer->consumer
9 }
```

Listing 5.1. TeeTime configuration DSL example

Listing 5.2 shows the generated Java source code. During generation, the class declaration is added in Line 4. The stage declaration and instantiation are split as shown in Line 6 and 7, respectively, in Line 11 and 12. Line 14 specifies the connection, and Line 16 defines the active declaration.

[1] https://www.eclipse.org/Xtext/

```
package teetime.test;
import teetime.stage.CollectorSink;
import teetime.stage.InitialElementProducer;
public class Config extends Configuration {
   // Stage instances
   private final InitialElementProducer producer;
   private final CollectorSink consumer;
   // Constructor
   public Config() {
      // Stage initialization
      producer = new InitialElementProducer();
      consumer = new CollectorSink();
      // Generated connections between all stages
      connectPorts(producer.getDefaultOutputPort(),
          consumer.getDefaultInputPort());
      // Declarations of active stages
      consumer.declareActive();
   }
}
```

Listing 5.2. Generated TeeTime configuration from DSL example

As mentioned in Section 4.2 only small changes to the configuration are necessary to transform it into a distributed configuration. The code does not specify its superclass. Instead, the DSL differs from non-distributed and distributed configurations via the file extension. A non-distributed configuration uses `.config` and a distributed one uses `.distributedconfig`. Furthermore, we added the `Node` statement in Line 8 and 9 to assign the stages to the nodes. In Line 8 the optional identifier *Producer* is specified. In the case of multiple stages per node, the stages can be one after another separated by a blank. Also, the transport protocol must be definable. It can be directly set on the connection arrow as shown in Line 11.

```
 1 import teetime.stage.InitialElementProducer
 2 import teetime.stage.CollectorSink
 3 Config(){
 4    // stages
 5    InitialElementProducer producer()
 6    CollectorSink consumer()
 7    // nodes
 8    Node(Producer) producer
 9    Node consumer
10    // connections
11    producer-TCP->consumer
}
```

Listing 5.3. TeeTime distributed configuration DSL example

Listing 5.4 shows the resulting source code generated by the DSL. Same as with the differences between a non-distributed and the distributed variant the corresponding DSL changes are small. The similarities support the transformation of non-distributed configuration into distributed ones written in the DSL.

```java
import teetime.stage.CollectorSink;
import teetime.stage.InitialElementProducer;
public class Config extends AbstractDistributedConfiguration {
   // Stage instances
   private final InitialElementProducer producer;
   private final CollectorSink consumer;
   // Constructor
   public Config() {
      // Stage initialization
      producer = new InitialElementProducer();
      consumer = new CollectorSink();
      // Node initialization
      createNodeForStages("Producer", producer);
      createNodeForStages(consumer);
      // Generated connections between all stages
      connectPorts(producer.getDefaultOutputPort(),
          consumer.getDefaultInputPort(), TransportProtocol.TCP);
   }
}
```

Listing 5.4. Generated distributed configuration

Chapter 6

Evaluation

In this chapter, we evaluate our implementation presented in Chapter 4 regarding feasibility and performance. We explain our evaluation goals in Section 6.1. All experiments share a common setup, which we describe in Section 6.2. We first present our results from the feasibility evaluation of the remote deployment, remote execution, and communication in Section 6.3. Afterward, we show our results from the fault handling tests in Section 6.4. After that, we describe the performance tests results in Section 6.5. In Section 6.6 we evaluate the extension of the DSL shown in Chapter 5.

6.1 Goals

The first purpose of the evaluation is to prove the feasibility of our approach to implementing a distributed Pipe-and-Filter architecture with TeeTime. Especially we want to evaluate the remote deployment, the remote execution, and the distributed communication. Therefore we implement multiple distributed configurations variants for all three transport protocols: TCP, UDP, and SSL over TCP as described in Section 6.2.2. The configurations contain only a few stages to focus on the distributed communication. We evaluate both socket communication as well as network communication separately by executing them locally on the same node as well as distributed on three different nodes. Additionally, we use the results to evaluate the communication overhead of network communication compared to local socket communication and JVM internal communication of non-distributed configurations.

The second goal is the feasibility of the fault tolerance. We generate different faults and test whether the application detects these faults and terminates gracefully.

As our third goal, we compare the performance of distributed configurations with their non-distributed variants to evaluate potential performance advantages. Therefore we use bigger configurations which can utilize more threads than one single node of our evaluation system can provide.

Finally we evaluate our DSL extension to support distributed configurations.

6.2 Common Experimental Setup

We perform our evaluation on the Cloud server of the Software Performance Engineering Lab (SPEL) from the Software Engineering research group at the Kiel University. We use three nodes of the eight Cloud nodes that the SPEL provides. Each node has the following specifications:

CPU	2x Intel Xeon E5-2650 (2.8GHz, 8 cores)
RAM	128 GB
OS	Ubuntu 14.04.5 LTS
Java	Java JDK 1.8. Update 121 64 bit

The nodes are connected via a 12x Fully Switched 10GBase-T network.

6.2.1 Stages

All test configurations as described in Section 6.2.2 are based on the following three stages:

RandomStringGeneratorStage

The `RandomStringGeneratorStage` is a producer stage, thus it has no input port and one output port. The stage constructor has two parameters: one for the string size and one to specify how many strings should be generated. The string is created by repetitively appending the char *a* until the string length equals the size defined by the string size parameter. Internally, Java requires 2 bytes per char. The size of one serialized char depends on the encoding and compression of the serializer. The stage generates a `TestObject` as shown in Listing 6.1 which contains the generated string and an index counter.

```java
package teetime.framework.distributed;
public class TestObject {
   int i;
   String j;
   public TestObject(final int i, final String j) {
      this.i = i;
      this.j = j;
   }
   public int getInt() {
      return i;
   }
   public String getString() {
      return j;
   }
}
```

Listing 6.1. `TestObject` class

CPULoadGeneratorStage

The `CPULoadGeneratorStage` has one input port and one output port. Rather than modifying the elements it receives, this stage withholds the element, generates CPU load for a specified amount of cycles and then forwards the element. A constructor parameter defines the cycle count. Thus it is variable for each stage instance. To generate the CPU load we use the `consumeCPU` method from the Blackhole benchmark provided by the JMH (Java Microbenchmark Harness[1]) framework. One cycle equates to approximately 3 CPU clocks depending on the processor architecture. The method ensures that the JVM cannot optimize the execution and therefore provides consistent execution times for each element.

EndStage

The `EndStage` has one input port and no output port. The stage has a counter attribute, which is incremented for every element received. When the stage receives an instance of a `TestObject` element. it prints the current counter attribute value and the index counter from the element to the console. Thus it is possible to verify whether the `EndStage` receives the elements in the same order as the `RandomStringGeneratorStage` created them.

6.2.2 TeeTime Configurations

We use four different configurations in our evaluation. The configurations are customizable. Thus we use each configuration in multiple setting variants. All configurations have constructor parameters to define the string size in bytes and the number of strings the `RandomStringGeneratorStage` produces. The distributed configurations have a further parameter to specify the transport protocol, which the `DistributedPipe` uses. A pipe may connect stages running in the same thread, in different threads or on different workers. Unsynchronized pipes connect stages running in the same thread, and synchronized pipes connect stages running in different threads, and distributed pipes connect stages running on different workers.

Non-Distributed Configuration

The `NonDistributedConfiguration` contains five stages as shown in Figure 6.1: One `RandomStringGeneratorStage`, three `CPULoadGeneratorStages` and one `EndStage`. The stages are executed on one single node in three threads to use the same amount of threads as the `DistributedConfiguration`. The cycle count of the `RandomStringGeneratorStage` is 3,000,000.

[1]http://openjdk.java.net/projects/code-tools/jmh/

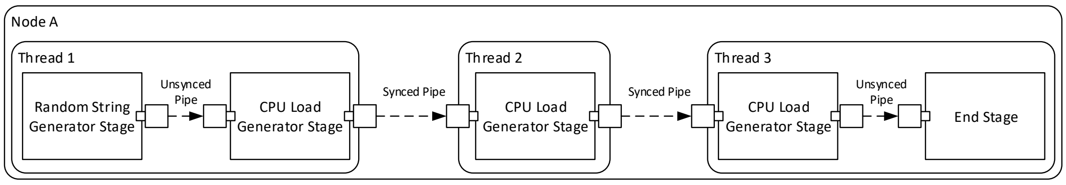


Figure 6.1. Non-distributed configuration with a cycle count of 3,000,000

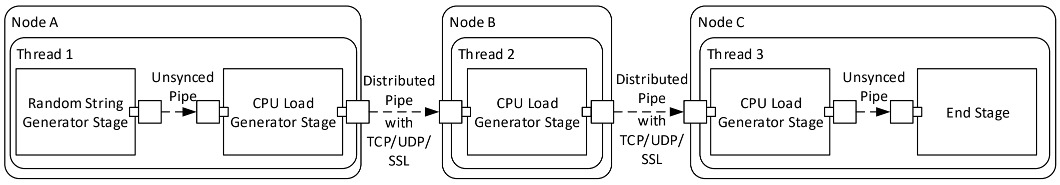


Figure 6.2. Distributed configuration with a cycle count of 3,000,000

The cycle count results in approximately three milliseconds of full load on one core with 2.8 GHz per element. The purpose of the configuration is the comparison with the distributed variant `DistributedConfiguration` to evaluate the communication overhead of the socket communication in Section 6.5.

Distributed Configuration

The `DistributedConfiguration` is very similar to the `NonDistributedConfiguration` and contains the same stages and thread assignment as shown in Figure 6.2. The distributed configuration requires three worker nodes for the execution instead of only one node such as the `NonDistributedConfiguration`. Each worker allocates one thread. This configuration is used to evaluate the remote deployment and remote execution of the configuration, as well as evaluating the distributed communication between the sender and receiver actors in Section 6.3. Furthermore, we use this configuration to evaluate the fault tolerance in Section 6.4 and the communication overhead in Section 6.5.

Big Non-Distributed Configuration

The `BigNonDistributedConfiguration` contains 48 stages, namely one `RandomStringGenerator-Stage`, 46 `CPULoadGeneratorStages` and one `EndStage`. These stages are connected with synchronized pipes as shown in Figure 6.3. Each thread executes exactly one stage. The cycle count of the `RandomStringGeneratorStage` is 30,000,000, which results in approximately 30 milliseconds of full load on one core with 2.8 GHz per element.

Big Distributed Configuration

The `BigDistributedConfiguration` is the distributed equivalent to the `BigNonDistributed-Configuration`. The stages are connected locally with synchronized pipes and between different nodes with distributed pipes as shown in Figure 6.4. The configuration requires three worker nodes for the execution. Each worker allocates 16 stages. We evaluate potential performance advantages with this configuration in Section 6.5.

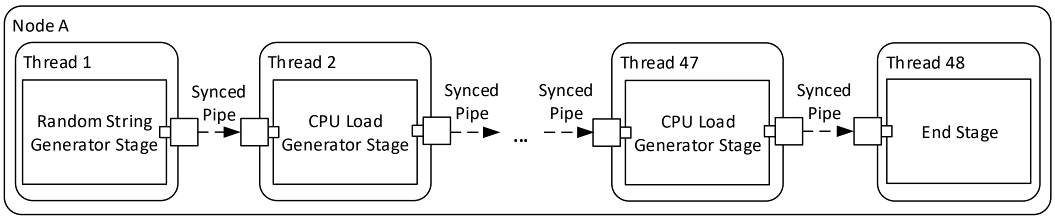


Figure 6.3. Big Non-Distributed configuration with a cycle count of 30,000,000

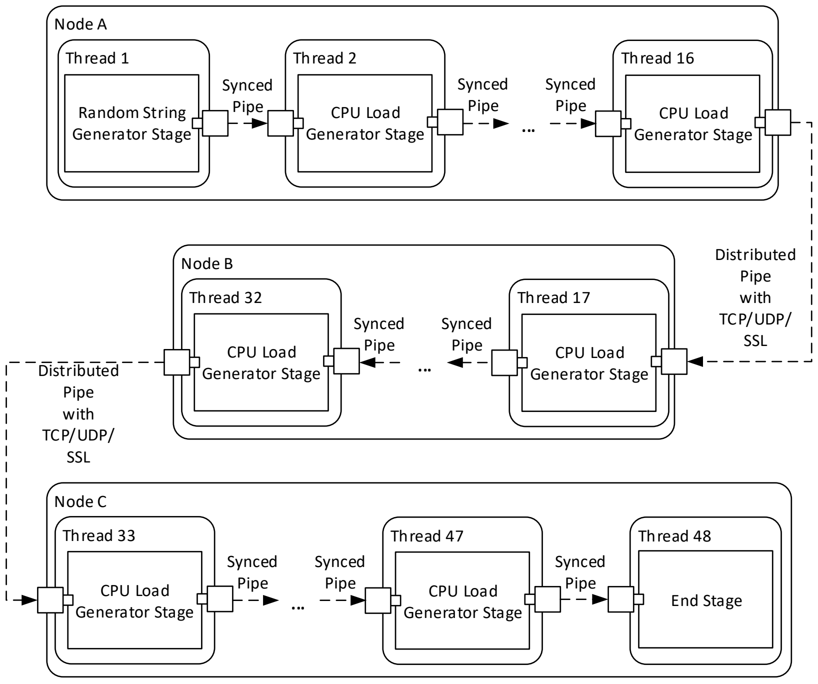


Figure 6.4. Big Distributed configuration with a cycle count of 30,000,000

6.2.3 Test Applications

We use a Java application named `Test` to execute the `DistributedConfiguration` and the `BigDistributedConfiguration`, as shown in Listing 6.2. To start the application, we need to parse four arguments: the name of the configuration to instantiate, a list of configuration constructor arguments, the host and the port of the master system. We extract the arguments in Line 9 to Line 12. Then we resolve the configuration class by name and instantiate the configuration with the constructor arguments in Line 13. Afterward, we execute the configuration in Line 14. Every second, the application checks whether the configuration is terminated. When the configuration terminates, the application terminates too.

```
 1 package teetime.distributed.test;
 2 import teetime.framework.distributed.AbstractDistributedConfiguration;
 3 public class Test {
 4    public static void main(String[] args) {
 5       if (args.length != 4) {
 6          System.out.println("Missing configuration arguments");
 7          return;
 8       }
 9       String configName = args[0];
10       String[] configArgs = args[1].split(",");
11       String host = args[2];
12       int port =  Integer.parseInt(args[3]);
13       AbstractDistributedConfiguration config =
            AbstractDistributedConfiguration.getConfig(configName, configArgs);
14       config.executeDistributed(host, port);
15       while(!config.isTerminated()){
16          System.out.println("Still executing...");
17          try {
18             Thread.sleep(1000);
19          } catch (InterruptedException e) {
20             e.printStackTrace();
21          }
22       }
23    }
24 }
```

Listing 6.2. Test application

To execute the non-distributed configurations `NonDistributedConfiguration` and `BigNon-DistributedConfiguration` we use the `NonDistributedTest` application shown in Listing 6.3. Similar to the `Test` application we parse configuration arguments in Line 13 to 15, then we instantiate the configuration in Line 19 or 23, depending on which of the two configurations should be used. Afterward, we create an `Execution` in Line 26 and execute it in Line 27.

```java
package teetime.distributed.test;
import teetime.distributed.test.configuration.BigNonDistributedConfiguration;
import teetime.distributed.test.configuration.NonDistributedConfiguration;
import teetime.framework.Configuration;
import teetime.framework.Execution;

public class NonDistributedTest {
   public static void main(String[] args) {
      if (args.length != 3) {
         System.out.println("Missing configuration arguments");
         return;
      }
      String configName = args[0];
      int size = Integer.parseInt(args[1]);
      int iterations = Integer.parseInt(args[2]);
      Configuration config;
      switch(configName){
         case "normal":
         config = new NonDistributedConfiguration(size, iterations);
         break;
         case "big":
         default:
         config = new BigNonDistributedConfiguration(size, iterations);
         break;
      }
      Execution<Configuration> execution = new Execution<Configuration>(config);
      execution.executeBlocking();
   }
}
```

Listing 6.3. Non-distributed test application

With this two applications and the `RemoteSystem` for the worker provided by TeeTime we execute all tests described in Section 6.3, Section 6.4 and Section 6.5.

6.3 Feasibility of Remote Deployment, Remote Execution and Distributed Communication

In this section, we evaluate the feasibility of the remote deployment of the configurations from the master to the worker nodes, the remote execution of the configuration on each worker and the distributed communication between the workers.

Table 6.1. Configuration Variants

Test #	Protocol	String Size	String Count	Communication Type
1	TCP	1	5000	Local
2	TCP	1048576	5000	Local
3	TCP	1	5000	Distributed
4	TCP	1048576	5000	Distributed
5	UDP	1	5000	Local
6	UDP	1048576	5000	Local
7	UDP	1	5000	Distributed
8	UDP	1048576	5000	Distributed
9	SSL over TCP	1	5000	Local
10	SSL over TCP	1048576	5000	Local
11	SSL over TCP	1	5000	Distributed
12	SSL over TCP	1048576	5000	Distributed

6.3.1 Methodology

We use the `DistributedConfiguration` described in Section 6.2.2 with the setting variants shown in Table 6.1. We test TCP, UDP and SSL in a local and a distributed setting with 5000 either small (a single char) or large (a string with 1,048,576 chars) objects. Each test is executed 20 times to test whether race conditions occur and to reuse the results in Section 6.5 for the communication overhead evaluation. In the local scenarios, we execute the master and all workers on the cloud node 6 (nc06). In the distributed scenarios we execute the master and Node A on cloud node 6 (nc06), Node B on cloud node 7 (nc07) and Node C on cloud node 8 (nc08). The worker logs relevant execution steps, e.g. the startup and termination of the worker system, the creation and termination of workers, the control messages between the master and workers, and the elements and signals sent between the workers. We use the log files to reconstruct the temporal sequence. Thus we can evaluate whether all steps executed correctly and in the right order and explain the output of the log reconstruction in Section 6.3.2.

6.3.2 Results & Discussion

An example run of the first test with the settings as listed in Section 6.2.2 is shown in Table 6.2. Initially, we create the actor systems. We create a master actor on the master node and a worker actor on each worker node. The order of the actor creation is irrelevant because the worker actors repeat trying to register to the master until they succeed. In this example the actor creation sequence started with the master actor, followed by the worker actor on worker 1, worker 3 and finally worker 2. Each worker registers to the master and receives a deploy message. Based on the message the worker fetch their configuration and instantiate the required sender and receiver actors. In this case worker 1 needed five times to resolve the corresponding receiver for its sender actor until the resolution is successful. The first four times between Line 11 and Line 16 are unsuccessful because worker 2 registers its receiver not until Line 29. As soon as all workers resolved all receivers, they notify the master that the senders are ready. After the last worker notifies the master in Line 38 that all senders are ready, the master sends a start signal to the worker to initiate the execution. All workers begin the execution of their configuration in Line 42. Thus the remote deployment and the remote execution works.

Furthermore, the distributed communication performs correctly, as we will show in the following. All signals and elements are transmitted from worker 1 to worker 2, which finally send it to worker 3. When worker 2 receives the `TerminatingSignal` in Line 58 it sends a poison pill to itself and notifies the corresponding sender to also shutdown. A poison pill is a shutdown message provided by Akka. The message inbox of the actor uses the FIFO principle. Thus the shutdown message is not processed until the actor processes all messages in the inbox. Hence the poison pill ensures a graceful shutdown. The sender also sends a poison pill to itself and starts the termination of the worker in Line 61. The worker waits for the termination of the `Execution` in Line 62 and then sends a poison pill to itself. The sender actor is the last running component in the pipeline on worker 1. Thus the execution of the node configuration was already terminated. If the pipeline contains branches, some stages may execute further after all sender actors terminated. Hence it is necessary to check if the execution is still running before terminating the worker completely. After the worker terminates, the whole remote system shuts down. Whenever the receiver actor on worker 3 receives the `TerminatingSignal`, it terminates itself. Since there are no sender actors on worker 3, no more receiver and sender actors are running. Thus the worker starts to terminate itself in Line 68. The stages following the receiver actor still run. Therefore the execution terminates not until Line 74. Finally, the worker sends a poison pill to itself, and the remote system shuts down.

Table 6.2. Example run of Test 1: TCP local with 5000 strings with a length of 1 char

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
1	15:48:52,317		Worker sys. created		
2	15:48:52,872			Worker sys. created	
3	15:48:53,327				Worker sys. created
4	15:48:53,844	Master sys. created			
5	15:48:53,848	Master actor created			

6	15:49:02,951		Worker Actor created		
7	15:49:03,417	Registered Node1			
8	15:49:03,421	Deployed Node1			
9	15:49:03,435		Deploy msg received		
10	15:49:03,435		Fetched Node1 config		
11	15:49:03,441		Receiver requested		
12	15:49:03,458		Execution created		
13	15:49:04,476		Receiver requested		
14	15:49:05,496		Receiver requested		
15	15:49:06,364				Created Worker Actor
16	15:49:06,516		Receiver requested		
17	15:49:06,585	RegisteredNode3			
18	15:49:06,585	Deployed Node3			
19	15:49:06,599				Deploy msg received
20	15:49:06,600				Fetched Node3 config
21	15:49:06,604				Registered receiver
22	15:49:06,616				Execution created
23	15:49:06,624	Waiting for more worker...			
24	15:49:06,878			Created Worker Actor	
25	15:49:07,102	Registered Node2			
26	15:49:07,103	Deployed Node2			
27	15:49:07,120			Deploy msg received	
28	15:49:07,121			Fetched Node2 config	
29	15:49:07,129			Receiver requested	
30	15:49:07,129			Registered receiver	
31	15:49:07,142				Receiver requested by Node2
32	15:49:07,143			Execution created	
33	15:49:07,163			Receiver ActorRef resolved	
34	15:49:07,164			All sender ready	
35	15:49:07,536		Receiver requested		

36	15:49:07,547			Receiver requested by Node1	
37	15:49:07,568		Receiver ActorRef resolved		
38	15:49:07,568		All sender ready		
39	15:49:07,571	Start signal to Node1			
40	15:49:07,572	Start signal to Node2			
41	15:49:07,572	Start signal to Node3			
42	15:49:07,574		Execution started	Execution started	Execution started
43	15:49:07,574		Signal sent		
44	15:49:07,590		Element sent		
45	15:49:07,595			Received start signal	
46	15:49:07,596			Signal sent	
47	15:49:07,602		Element sent		
48	15:49:07,603			Received element	
49	15:49:07,605			Received element	
50	15:49:07,628			Element sent	Received start signal
51	15:49:07,632				Received element
52	15:49:07,640			Element sent	
53	15:49:07,642				Received element
54	...	...	...	...	...
55	15:49:51,739		Element sent		
56	15:49:51,739			Received element	
57	15:49:51,740		Signal sent	Element sent	
58	15:49:51,743			Received TerminatingSignal	
59	15:49:51,744			Poison pill (Receiver)	
60	15:49:51,747		Poison pill (Sender)		
61	15:49:51,748		Terminate worker		
62	15:49:51,749		Execution terminated	Element sent	
63	15:49:51,749		Poison pill (Worker)		Received element
64	15:49:51,749			Signal sent	
65	15:49:51,750	Worker Node1 ack shutdown			
66	15:49:51,752				Received TerminatingSignal
67	15:49:51,753				Poison pill (Receiver)

68	15:49:51,754				Terminate worker
69	15:49:51,756			Poison pill (Sender)	
70	15:49:51,757			Terminate worker	
71	15:49:51,757			Execution terminated	
72	15:49:51,757			Poison pill (Worker)	
73	15:49:51,758	Worker Node2 ack shutdown			
74	15:49:53,488				Execution terminated
75	15:49:53,488				Poison pill (Worker)
76	15:49:53,489	Worker Node3 ack shutdown			
77	15:49:53,490	Poison pill			
78	15:49:56,874				Shutdown system
79	15:49:56,876		Shutdown system		
80	15:49:56,879			Shutdown system	
81	15:49:56,881	Shutdown system			

This example run shows the successful remote deployment and remote execution, a working distributed communication between the worker and a clean shutdown of the system after the execution completes. The distributed communication only starts when the remote deployment and remote execution succeeded. Test 2 to 12 from Section 6.2.2 only differ from Test 1 in the distributed communication and string size. Thus we omit the startup procedure, which Table 6.2 already presents and concentrate on the results of the distributed communication for these tests. In Table 6.3 we show that the distributed communication also works for large objects. All distributed communication experiments succeeded. We show the remaining results for experiment 3 to 12 in the appendix in Table A.1 to Table A.10 since they differ only in the timestamps and in a few cases in the temporal order the receiver actor receives the starting signal and the sender actor sends the first element.

6.3.3 Threats to Validity

We use strings to test the communication. These strings are embedded into a `TestObject` to use a custom class which does not implement the `Serializable` interface. A `TestObject` contains the string and an integer as an index counter. This object consists only of two primitive attributes. More complex objects like for example an object with an object reference cycle to itself (an instance A of an object O1, which contains an instance of object O2 as an attribute, which has instance A of object O1 as an attribute) may be problematic. It may be possible that Kryo can not serialize these objects. In this case, the distributed communication would not work.

The distributed communication is tested for socket communication on the same node and between nodes in the same network. We did not evaluate the distributed communication across different local networks or the Internet because all nodes of the Software Performance Engineering Lab are on the same network and are not accessible through the Internet.

6.4 Feasibility of Fault Tolerance

In general, faults may occur either in the Java application or its environment. Inside the application, the fault source is either in Akka or TeeTime. Faults which occur in the custom code of the user are also covered by TeeTime because the code is executed inside the TeeTime stages, thus are caught by TeeTime. Faults in the environment are crashes of the JVM, or the whole Node, or the network. A network failure would lead to a network partition. We evaluate these types of faults in this section. For the Akka and TeeTime faults, we throw exceptions during the execution. To evaluate the death of a master or a worker we kill the corresponding processes. Whether the node becomes unreachable due to a crash or a network partition cannot be determined, such that the fault handling mechanism is the same. In the case of a network partition, the nodes in each partition must handle the fault independently because the master can only be part of one partition. Thus the other partitions can not be controlled by the master anymore.

6.4.1 Methodology

We again use the `DistributedConfiguration` described in Section 6.2.2. The actors always send the control messages over TCP. Thus the different transport protocol variants do not have to be tested separately.

Table 6.3. Distributed communication of Test 2: TCP local, 5000 strings with a length of 1,048,576

Timestamp	Worker 1	Worker 2	Worker 3
	Startup		
16:10:55,461	Signal sent		
16:10:55,481	Element sent		
16:10:55,482		Received signal	
16:10:55,483		Signal sent	
16:10:55,488		Received element	
16:10:55,502			Received signal
16:10:55,504		Element sent	
16:10:55,508			Received element
...	...	...	...

To examine the fault handling of TeeTime and Akka Exceptions, we configure the `DistributedConfiguration` with 1000 strings with a size of 1 char. For the master and worker death we use the same configuration but with 2000 strings to have enough time to interrupt the execution manually. We execute each of the following tests once. We run the master and worker 1 on cloud node 6 (nc06), worker 2 on cloud node 7 (nc07) and worker 3 on cloud node 8 (nc08).

TeeTime Exception

First, we test the handling of TeeTime exceptions. Therefore we enable the TeeTime exception test in the configuration. A TeeTime exception is thrown inside the `RandomStringGeneratorStage` after 500 strings were processed.

Akka Exception

After the TeeTime exception, we test the handling of Akka exceptions. Therefore we enable the Akka exception test in the configuration. We replace the generated string in the `RandomStringGeneratorStage` after 500 strings were processed with the string *fail*. When the first sender actor recognize the *fail* string, it throws an `IllegalStateException`.

Master / Worker death

We test the handling of master and worker death manually. We execute the master and each of the three workers with a shell script in the terminal, which waits for any key to be pressed after executing the Java applications. If a key is pressed, the script kills the Java application and saves the timestamp of the kill command to a log file. Thus we can test both master and worker death independently by killing the corresponding Java application in the terminal.

Network partition

We do not test the network partition separately because the master and worker death handling tests already cover this scenario. A master death is similar to a network partition where the workers are separated from the master because the workers cannot communicate with the master in both cases and won't receive any control messages. A worker death is similar to a network partition where a worker is separated from the master and the other workers because the master and the remaining workers cannot communicate with the worker anymore. Thus it makes no difference for the fault detection if we stop the network between the nodes or killing the application on the nodes.

6.4.2 Results & Discussion

The TeeTime exception test is shown in Table 6.4. After 500 strings were processed, the `IllegalStateException` is thrown on worker 1 in the `RandomStringGeneratorStage`. The supervisor of the worker actor catches the exception in Line 1. The worker notifies the master about the occurred error. After that, it starts its shutdown procedure. The master receives the error message in Line 2. Meanwhile, the master logs the error message and sends shutdown signals to the remaining workers. Each worker acknowledges the start of its shutdown procedure and aborts its TeeTime execution. Shortly after the TeeTime execution aborts, the workers shutdown their actor systems and terminate. After all workers acknowledged the shutdown the master also terminates.

Table 6.4. TeeTime exception test run

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
Distributed configuration running normally...					
1	17:20:43,323		**TeeTime crashed**		
2	17:20:43,343	**Error on Worker**			
3	17:20:43,348	Send shutdown signal to worker 3			
4	17:20:43,348	Send shutdown signal to worker 2			
5	17:20:43,354	Worker 3 acknowledged shutdown			
6	17:20:43,355	Worker 2 acknowledged shutdown			
7	17:20:43,951			Execution aborted	
8	17:20:44,362				Execution aborted
9	17:20:45,225		Shutdown Actor system		
10	17:20:45,230	Shutdown Actor system			
11	17:20:45,822			Shutdown Actor system	
12	17:20:46,234				Shutdown Actor system

The Akka exception test is shown in Table 6.5. After 500 strings were processed the `RandomStringGeneratorStage` generates the *fail* string which the sender actor recognize, thus it throws an `IllegalStateException`. The supervisor of the worker actor catches the exception in Line 1. The worker aborts the TeeTime execution in Line 2 and notifies the master about the occurred error. The master receives the error message in Line 3. Just like the TeeTime exception handling, the master sends shutdown signals to the remaining workers, waits for the shutdown acknowledgments and terminates itself. After receiving the shutdown signals, the workers abort the execution and terminate.

Table 6.5. Akka exception test run

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
		Distributed configuration running normally...			
1	17:20:07,434		**Actor crashed**		
2	17:20:07,441		Execution aborted		
3	17:20:07,464	**Error on Worker**			
4	17:20:07,471	Send shutdown signal to worker 3			
5	17:20:07,471	Send shutdown signal to worker 2			
6	17:20:07,482	Worker 2 acknowledged shutdown			
7	17:20:07,505	Worker 3 acknowledged shutdown			
8	17:20:08,076			Execution aborted	
9	17:20:08,489				Execution aborted
10	17:20:09,587		Shutdown Actor system		
11	17:20:09,593	Shutdown Actor system			
12	17:20:10,186			Shutdown Actor system	
13	17:20:10,601				Shutdown Actor system

We show the master death handling test in Table 6.6. We kill the master in Line 1. In Line 2 worker 1 decides that the master node is permanently not available anymore, aborts its TeeTime execution, and shuts down. In Line 5 also worker 3 decides that the master node is not available anymore, also aborts its execution, and shuts down. The sender actor of worker 2 is the supervisor of the receiver actor of worker 3. Due to the termination of worker 3, the sender actor is notified and also starts its shutdown procedure. Note that even without this notification, worker 2 would recognize that the master is not available anymore, but this would require more time until its termination. The supervision of the receiver helps to avoid sending messages to a remote actor who is no longer available.

Table 6.7 shows the worker death handling test. We kill Worker 1 in Line 1. The master decides approximately 16 seconds later that the worker is unreachable in Line 2. In the case of a worker death, the master handles the error completely. It sends shutdown signals to worker 2 and 3 and shuts down after receiving acknowledgments from both workers. The workers abort the TeeTime execution and also terminate.

Table 6.6. Master death test run

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
Distributed configuration running normally...					
1	17:29:12,000	**Master crashed**			
2	17:29:29,083		**Master terminated**		
3	17:29:29,090		Execution aborted		
4	17:29:30,090		Shutdown Actor system		
5	17:29:30,119				**Master terminated**
6	17:29:30,125				Execution aborted
7	17:29:30,223			**Monitored worker 3 terminated**	
8	17:29:30,231			Execution aborted	
9	17:29:31,105				Shutdown Actor system
10	17:29:31,195			Shutdown Actor system	

Table 6.7. Worker death test run

LN	Timestamp	Master	Worker 1	Worker 2	Worker 3
Distributed configuration running normally...					
1	17:26:28,355		**Node1 crashed**		
2	17:26:44,626	**Worker 1 terminated unexpected**			
3	17:26:44,627	Send shutdown signal to worker 3			
4	17:26:44,627	Send shutdown signal to worker 2			
5	17:26:44,633	Worker 3 acknowledged shutdown			
6	17:26:44,635	Worker 2 acknowledged shutdown			
7	17:26:45,612	Shutdown Actor system			
8	17:26:45,641				Execution aborted

9	17:26:46,232			Execution aborted	
10	17:26:46,624				Shutdown Actor system
11	17:26:47,214			Shutdown Actor system	

As shown in Table 6.4, Table 6.5, Table 6.6 and Table 6.7 the fault handling works as desired: The application handles all faults correctly. In the case of Akka and TeeTime, the worker can notify the master about the fault. When a node becomes unavailable the system first waits for the node to become available again, before it decides that the node is permanently unavailable. Thus the fault handling starts quicker for application internal errors than that for unreachable nodes.

6.4.3 Threats to Validity

The unavailability of a node can be temporary, prohibiting an exact identification of a node crash. Thus we can only use a timeout to determine if a node is unreachable. There is a chance that a node is falsely marked as permanently unreachable and the system terminates although the node is still available. The auto-downing after a given period is maybe too rigid to avoid false fault recognition in case of network partitions. Moreover, the fault handling in case of a network partition was not tested separately based on the assumption that the node death handling is equivalent.

We only tested the fault tolerance after the cluster was built up. The start-up phase has not been evaluated yet. Furthermore, no Akka exception handling has been tested inside the master actor.

6.5 Performance

In this section, we evaluate the performance overhead of the distributed communication and the potential reduction of the execution time.

6.5.1 Methodology

We use the `DistributedConfiguration` to test the communication overhead with the same setting variants as in Section 6.3, reusing the logs generated by the previous test runs. For comparison, we use the `NonDistributedConfiguration` executed 20 times with 5000 strings, once with a size of 1 char and once with a size of 1,048,576 chars.

To evaluate potential performance advantages, we use the `BigDistributedConfiguration` as described in Section 6.2.2. We execute the configuration with the setting variants shown in Table 6.8, which are identical to the settings described in Table 6.1 used for the

Table 6.8. Configuration Variants

Test #	Protocol	String Size	String Count	Communication Type
1	TCP	1	5000	Local
2	TCP	1048576	5000	Local
3	TCP	1	5000	Distributed
4	TCP	1048576	5000	Distributed
5	UDP	1	5000	Local
6	UDP	1048576	5000	Local
7	UDP	1	5000	Distributed
8	UDP	1048576	5000	Distributed
9	SSL over TCP	1	5000	Local
10	SSL over TCP	1048576	5000	Local
11	SSL over TCP	1	5000	Distributed
12	SSL over TCP	1048576	5000	Distributed

`DistributedConfiguration`. Each variant is executed 20 times. For comparison, we use the `BigNonDistributedConfiguration` executed 20 times with 5000 strings, once with a size of 1 char and once with a size of 1,048,576 chars.

We calculate the average execution time of each variant and the corresponding confidence interval of 95% for both tests.

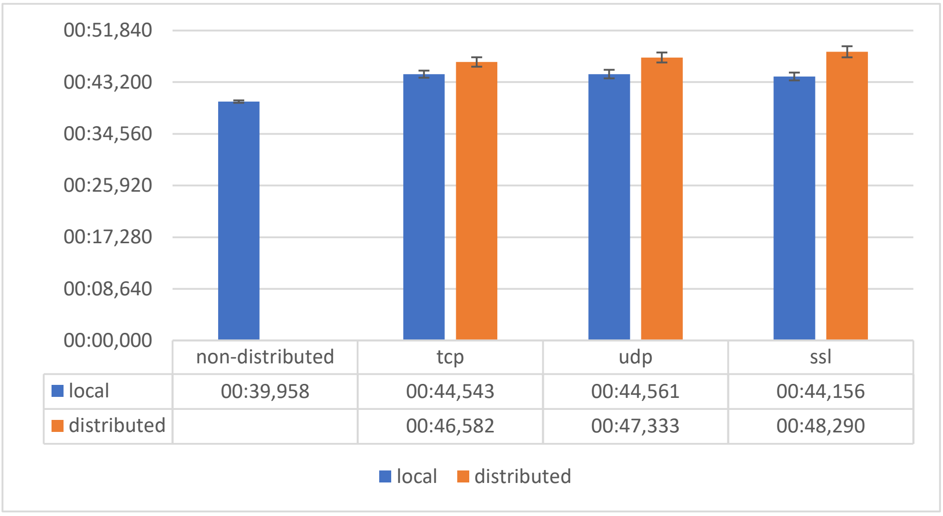
6.5.2 Results & Discussion

The average execution times for the `DistributedConfiguration` with small objects are shown in Table 6.9 and visualized in Figure 6.5. The execution time for the non-distributed TeeTime configuration is 00:39,958, determined with the `NonDistributedConfiguration`. The overhead for local socket communication over TCP, UDP or SSL is approximately 4.5 seconds in this scenario. The confidence interval is between ±00:00,205 and ±00:00,715, thus the execution times over the socket are significantly higher, but there is no clear difference between TCP, UDP or SSL. In the case of the distributed communication TCP, UDP and SSL the execution times are about 3 seconds longer than the local socket communication and more than 7 seconds longer than the non-distributed variant. Considering the confidence interval TCP, UDP, and SSL do not differ significantly.

The average execution times for the `DistributedConfiguration` with small objects are shown in Table 6.10 and visualized in Figure 6.6. The execution time for the non-distributed TeeTime configuration is 00:40,313 ±00:00,306, hence the difference to the small objects

Table 6.9. Average execution times for the `DistributedConfiguration` with small objects

Protocol	Local	CI 95%	Distributed	CI 95%
Non-distributed	00:39,958	00:00,205		
TCP	00:44,543	00:00,595	00:46,582	00:00,790
UDP	00:44,561	00:00,715	00:47,333	00:00,835
SSL over TCP	00:44,156	00:00,653	00:48,290	00:00,927



	non-distributed	tcp	udp	ssl
🟦 local	00:39,958	00:44,543	00:44,561	00:44,156
🟧 distributed		00:46,582	00:47,333	00:48,290

Figure 6.5. `DistributedConfiguration` with small objects

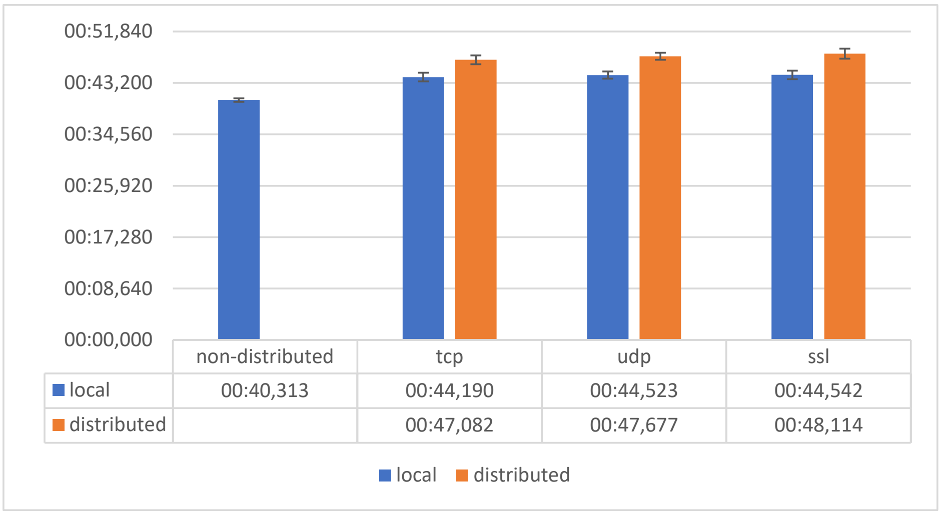
with 00:39,958 ±00:00,205 is not significant. The overhead of local communication is approximately 4 seconds with a confidence interval greater than ±00:00,606 thus with no clear difference to the small objects overhead. The overhead of the distributed communication is on average 7.5 seconds which does not significantly differ from the distributed communication with small objects.

Thus the message size has no clear impact on the execution time for this pipeline configuration. The influence of a slower network than the 1Gbit/s local network connection remains to be evaluated. A connection between different networks, especially over the Internet, may increase the impact of the message size. Though the distributed communication clearly decreases the execution times compared to the non-distributed communication in the scenario tested here.

To evaluate the performance advantages, we use the `BigDistributedConfiguration`. The results of the average execution times for small objects are shown in Table 6.11 and

Table 6.10. Average execution times for the `DistributedConfiguration` with large objects

Protocol	Local	CI 95%	Distributed	CI 95%
Non-distributed	00:40,313	00:00,306		
TCP	00:44,190	00:00,724	00:47,082	00:00,742
UDP	00:44,523	00:00,606	00:47,677	00:00,590
SSL over TCP	00:44,542	00:00,716	00:48,114	00:00,840

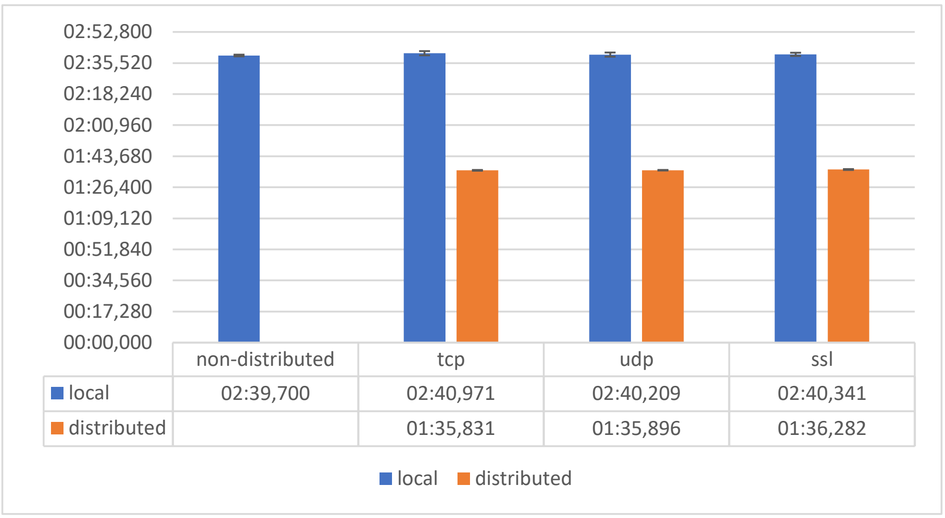


	non-distributed	tcp	udp	ssl
🟦 local	00:40,313	00:44,190	00:44,523	00:44,542
🟧 distributed		00:47,082	00:47,677	00:48,114

Figure 6.6. `DistributedConfiguration` with large objects

visualized in Figure 6.7 and the results for large objects are shown in Table 6.12 and visualized in Figure 6.8.

Equivalent to the `DistributedConfiguration`, the message size has no impact on the execution times. Thus the results exhibit no significant difference between small and large objects. Furthermore, the local communication with TCP, UDP, and SSL do not clearly differ from the non-distributed communication. The higher CPU load per stage decreases the impact of the distributed communication because the calculation on an element takes longer than the network transmission of the element to the next stage. The CPU on the cloud node provides 2x 8 cores and is thus able to execute the 16 stages per worker each in its own thread when we execute the configuration in a distributed setting on different nodes. In the local setting, 48 stages have to share the 16 threads. As a result, the parallelism increases in the distributed variant, so we gain a reduction of the execution time by approximately 40%. Additionally, the confidence interval is clearly smaller in the distributed setting, indicating



	non-distributed	tcp	udp	ssl
■ local	02:39,700	02:40,971	02:40,209	02:40,341
■ distributed		01:35,831	01:35,896	01:36,282

Figure 6.7. `BigDistributedConfiguration` with small objects

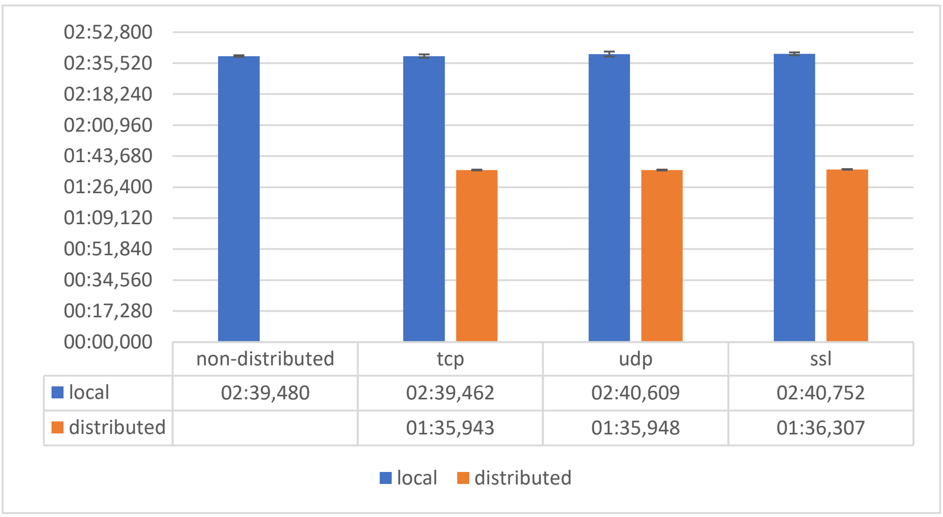
Table 6.11. Average execution times for the `BigDistributedConfiguration` with small objects

Local	CI 95%	Distributed	CI 95%
02:39,700	00:00,483		
02:40,971	00:01,182	01:35,831	00:00,221
02:40,209	00:01,165	01:35,896	00:00,168
02:40,341	00:00,898	01:36,282	00:00,248

that the execution time is more stable. The reduced impact of the thread scheduling due to fewer threads per worker node could be a reason for this behavior. The performance advantage is limited with this linear pipeline configuration. An element has to be fully processed by the first worker before the second worker can start to process this element, which reduces the level of parallelism.

6.5.3 Threats to Validity

We use the logs to calculate the execution times. In the local setting, all worker run on the same node and thus use the same system clock. In the case of the distributed setting, the workers run on different nodes with different system times. We try to eliminate the time differences by subtracting approximately 10-11 seconds for cloud node 7 and 6-7 seconds for cloud node 8. We gain the time differences by executing SSH scripts on each worker to



	non-distributed	tcp	udp	ssl
🟦 local	02:39,480	02:39,462	02:40,609	02:40,752
🟧 distributed		01:35,943	01:35,948	01:36,307

Figure 6.8. `BigDistributedConfiguration` with large objects

Table 6.12. Average execution times for the `BigDistributedConfiguration` with large objects

Local	CI 95%	Distributed	CI 95%
02:39,480	00:00,463		
02:39,462	00:00,959	01:35,943	00:00,233
02:40,609	00:01,386	01:35,948	00:00,250
02:40,752	00:00,803	01:36,307	00:00,183

get the system time of the other nodes multiple times. Due to the SSH connection build up, the results vary by about one second.

The performance advantage of 40% applies to our specific configuration scenario. Different configurations have a different level of parallelization capability. Thus the advantage can result in a higher or lower reduction in the execution time. Furthermore, the environment in which the configuration is executed, especially the speed of the network and the CPU performance of the nodes, have a further impact on the application performance.

6.6 Support of Distributed Configurations in the TeeTime DSL

6.6.1 Methodology

We execute the Distributed DSL project by running it as an Eclipse Application from inside Eclipse. In the editor, we create a file named `Example.distributedconfig`, which contains the code shown in Listing 5.3, and save it.

6.6.2 Results & Discussion

The DSL plug-in automatically generates the Java source on files with the `.distributedconfig` file type on save. The output matches the code shown in Listing 5.4, thus the DSL works as desired.

6.6.3 Threads to Validity

We evaluated the correctness of the DSL only for this single example. We did this by comparing the desired output with the generated output. Therefore we assume that the DSL provided by Zloch and de Sousa works correctly thus only the changes mentioned in Chapter 5 were required. It may be possible that configurations exist, which the DSL can not describe with the current grammar.

Chapter 7

Related Work

In this chapter, we compare our implementation approach of a distributed Pipe-and-Filter architecture with related work.

We used the actor framework Akka to implement our distributed pipe in TeeTime. Akka provides actors who can communicate both with local and remote actors transparently. As a result, it would also be possible to build an actor based distributed Pipe-and-Filter architecture completely with Akka. TeeTime stages can have multiple input and output ports, hence can contain branches and loops. In contrast, actors only have one input port: their mailboxes. An actor mailbox is untyped. Thus it can receive various messages from different actors. Actors can also send messages to each actor in the same cluster. Consequently, an pipeline based on actors can have branches and loops. In contrast to the typed TeeTime ports, an actor needs to check and type cast each message. Therefore message types which cannot be handled by an actor are only detected at runtime [Wulf et al. 2017]. Faulty connections are directly discoverable in TeeTime during the configuration, since pipes can only connect ports of the same type. The type safety applies as well to the distributed pipe based on the sender and receiver actors as described in Section 4.1.1, because the distributed pipe guarantees that only ports of the same type are connected. Furthermore the objects are wrapped into either a `TeeTimeDataMessage` or `TeeTimeSignalMessage`, which the receiver actor has type checks for. Additionally, TeeTime provides with the configuration a central component to build the architecture and as a result an overview of the entire architecture. We are not aware of a central component like the TeeTime configuration provided by Akka to connect actors to build the architecture.

FastFlow is a parallel processing framework for C++ which support the Pipe-and-Filter architectural style. The main project does not support distributed pipelines. Aldinucci et al. proposed a FastFlow extension to support distributed systems [Aldinucci et al. 2012]. Equivalent to TeeTime FastFlow uses message communication between the nodes instead of shared memory or remote procedure calls. FastFlow only supports one input and one output port per stage. As a further restriction, one stage can either have one distributed input port or one distributed output port, but not both at the same time. Thus FastFlow does not support nodes with only one stage. Furthermore, the support for branches is very restricted by always distributing each element to each subsequent stage.

Apache Spark Streaming as part of Apache Spark (Section 3.2.4) and Apache Storm (Section 3.2.5) are two Pipe-and-Filter frameworks, which provide distributed computing. Same as TeeTime Apache Storm uses Kryo for serialization. Spark uses a distributed filesystem, for example, Hadoop Distributed File System (HDFS) instead of direct message communication for the data. If Spark needs to serialize messages it also provides Kryo as a serializer but uses the Java serialization as default. Both Spark and Storm focus on streaming and lack the support for more than one input and output port per stage. Storm uses a directed acyclic graph to define its topology. Thus loops are not possible.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we introduced and implemented an approach to extend TeeTime to support distributed Pipe-and-Filter architectures. After evaluating several Java frameworks for distributed systems, we chose the Akka framework. Based on our architecture approach we successfully implemented different example distributed Pipe-and-Filter architectures by integrating Akka into TeeTime.

We provide a single configuration to specify the distributed architecture. Thus the user has an overview of the entire architecture, especially of the connections between stages running on different nodes. Hence the user does not need to handle multiple configurations for a single architecture at the same time. Additionally, he can easily relocate stages between worker nodes by simply changing the node assignment of the stage without switching between configurations. The distributed configuration is very similar to the non-distributed variant. Therefore users who are familiar with TeeTime will be able to transform existing non-distributed configurations into distributed ones with low effort. It is enough to change the base class of the configuration and to assign each stage to a node.

Furthermore, TeeTime generates configurations for each node from the distributed configuration, automatically deploys them to the worker nodes and subsequently executes them. The user can either specify the node a configuration should be deployed to or let TeeTime automatically choose a node. Thus the user can assign stages to generic node configurations without the need to exactly specify the particular worker nodes the configurations should be executed on. Currently, the execution of the remote systems still requires a manual deployment of the application's JAR file in conjunction with the TeeTime library, so that all required class files are available.

We implemented a distributed pipe to connect stages on different nodes. The distributed communication relies on two Akka actors: one sender actor and one receiver actor. For serialization, we use the Kryo serializer. Kryo is faster than the standard Java serializer and is used by other Pipe-and-Filter frameworks like Apache Storm as well. As shown in Section 6.3 the distributed communication works successfully.

Moreover, we provide basic fault tolerance capabilities. TeeTime can detect faults inside the application and in its environment, e.g. network faults, as shown in Section 6.4. The desired fault tolerance behavior would be to recover the system so that the execution can resume after a fault occurs. Fault recovery in Pipe-and-Filter architectures is complex due to a lot of concurrent states which must be recovered at the same moment which is even more difficult in a distributed scenario as explained in Section 4.4. Thus we provide a graceful termination of the entire distributed system in case of a fault.

Our motivation for a distributed Pipe-and-Filter architecture were performance enhancements and data locality. In our distributed evaluation scenarios in Section 6.5 we reduced the execution time by approximately 40% compared with the non-distributed configuration. Thus we reached our goal. We proved the performance enhancements for our specific evaluation scenarios, whereby we can not guarantee performance improvements in general. Other scenarios may provide lower, higher, or no enhancements, or in the worst case a performance decrease due to the network overhead depending on the underlying architecture. Furthermore, we provide data locality on systems which support Java and are accessible via the network. Thus the systems needs to be able to execute the TeeTime remote system and able to join the actor cluster.

Finally, we also provide a DSL to simplify the declaration of distributed configurations. Thus the user can avoid writing much boilerplate code. Consequently, he gets a simpler and clearer overview of the Pipe-and-Filter architecture which he describes via the configuration.

Overall we reached all our goals and successfully evaluated our implementation.

8.2 Future Work

In this section, we present future work, especially implementation ideas to further improve the distributed capabilities of TeeTime.

Compared with non-distributed Pipe-and-Filter architectures the distributed ones can provide performance improvements and further capabilities through data locality, like faster data access or data access at all. The use of the distributed configurations also has limitations compared to the non-distributed variants, and it is not possible to transform every non-distributed configurations to a distributed one. The access to stage attributes after or during the execution is currently not possible. Consequently, non-distributed configurations which need access to stage attributes cannot be converted to a distributed configuration. Due to that, we want to enable the access to stage attributes and thus the possibility to transform these configurations into distributed ones.

Currently, we focused our evaluation on the distributed communication with nodes running on the same network. We want to evaluate whether the nodes can create and join the cluster when they are executed on different networks, which the Internet connects. Furthermore, we want to evaluate whether the distributed pipe is in working order when used via the Internet and how the performance behaves in this scenario.

When the communication between the nodes relies on a potentially insecure network like the Internet, we provide encryption for the distributed pipe. Thus the user can encrypt data and signals transmitted over the pipe. The control sequences between the master and the workers are currently sent over plain TCP and do not support encryption. Thus control messages can be intercepted and manipulated by an attacker. We want to provide encryption also for the control messages to prevent these man-in-the-middle attacks.

Furthermore, the cluster does not require an authentication. Thus a new node can simply join the cluster and send malicious messages to other cluster members. Besides the encryption of the control messages, we also want to implement an authentication to prevent this kind of attacks and consequently to improve the security.

TeeTime supports the direct execution of non-distributed configurations with the TeeTime library without embedding the configuration in a Java application. Currently, we provide remote systems to execute the worker nodes. The user must embed the master node into a Java application. Equivalent to the non-distributed configuration we want to provide a master system as a counterpart to the remote system, which executes the distributed configuration. In case the user utilizes the master application only to execute the distributed configuration with no additional logic, he can instead use this master system. Thus the user only has to implement its distributed configuration and can execute it through the TeeTime library.

The automatic configuration deployment and execution still requires the manual start of the master application and on each node the start of a remote system. Therefore the TeeTime and configuration class files must be deployed to the nodes. We want to provide a ssh script which can be configured with the name of the Pipe-and-Filter configuration and a list of nodes, where to deploy the configurations. The scripts should copy the JAR files to the nodes and execute them. Thus the entire distributed Pipe-and-Filter architecture could be deployed and executed through this script.

Appendix A

Distributed Communication Test Results

Table A.1. Distributed communication of Test 3: TCP distributed, 5000 strings with a length of 1

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
16:32:36,879	Signal sent		
16:32:36,899	Element sent		
16:32:38,047		Received signal	
16:32:38,048		Signal sent	
16:32:38,054		Received element	
16:32:38,075		Element sent	
16:32:38,698			Received signal
16:32:38,703			Received element
...	...	...	...

Table A.2. Distributed communication of Test 4: TCP distributed, 5000 strings with a length of 1,048,576

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
16:55:34,091	Signal sent		
16:55:34,114	Element sent		
16:55:35,263		Received signal	
16:55:35,264		Signal sent	
16:55:35,273		Received element	
16:55:35,292		Element sent	
16:55:35,895			Received signal
16:55:35,907			Received element
...	...	...	...

Table A.3. Distributed communication of Test 5: UDP local, 5000 strings with a length of 1

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
17:18:44,216	Signal sent		
17:18:44,233	Element sent		
17:18:44,240		Received signal	
17:18:44,241		Signal sent	
17:18:44,252		Received element	
17:18:44,272		Element sent	
17:18:44,280			Received signal
17:18:44,283			Received element
...	...	...	...

Table A.4. Distributed communication of Test 6: UDP local, 5000 strings with a length of 1,048,576

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
17:40:22,478	Signal sent		
17:40:22,499	Element sent		
17:40:22,507		Received signal	
17:40:22,508		Signal sent	
17:40:22,514		Received element	
17:40:22,531		Element sent	
17:40:22,536			Received signal
17:40:22,539			Received element
...	...	...	...

Table A.5. Distributed communication of Test 7: UDP distributed, 5000 strings with a length of 1

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
18:01:58,175	Signal sent		
18:01:58,198	Element sent		
18:01:59,373		Received signal	
18:01:59,374		Signal sent	
18:01:59,380		Received element	
18:01:59,400		Element sent	
18:02:00,012			Received signal
18:02:00,016			Received element
...	...	...	...

Table A.6. Distributed communication of Test 8: UDP distributed, 5000 strings with a length of 1,048,576

Timestamp	Worker 1	Worker 2	Worker 3
Startup			
18:25:20,293	Signal sent		
18:25:20,317	Element sent		
18:25:21,503		Received signal	
18:25:21,505		Signal sent	
18:25:21,516		Received element	
18:25:21,544		Element sent	
18:25:22,146			Received signal
18:25:22,150			Received element
...	...	...	...

Table A.7. Distributed communication of Test 9: SSL local, 5000 strings with a length of 1

Timestamp	Worker 1	Worker 2	Worker 3
Startup			
18:48:55,398	Signal sent		
18:48:55,413	Element sent		
18:48:55,414		Received signal	
18:48:55,415		Signal sent	
18:48:55,429		Received element	
18:48:55,438			Received signal
18:48:55,444		Element sent	
18:48:55,450			Received element
...	...	...	...

Table A.8. Distributed communication of Test 10: SSL local, 5000 strings with a length of 1,048,576

Timestamp	Worker 1	Worker 2	Worker 3
Startup			
19:33:11,706	Signal sent		
19:33:11,718	Element sent		
19:33:11,721		Received signal	
19:33:11,724		Signal sent	
19:33:11,759		Received element	
19:33:11,759		Element sent	
19:33:11,780			Received signal
19:33:11,784			Received element
...	...	...	...

Table A.9. Distributed communication of Test 11: SSL distributed, 5000 strings with a length of 1

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
20:15:33,685	Signal sent		
20:15:33,709	Element sent		
20:15:34,923		Received signal	
20:15:34,924		Signal sent	
20:15:34,929		Received element	
20:15:34,951		Element sent	
20:15:35,529			Received signal
20:15:35,536			Received element
...	...	...	...

Table A.10. Distributed communication of Test 12: SSL distributed, 5000 strings with a length of 1,048,576

Timestamp	Worker 1	Worker 2	Worker 3
		Startup	
20:39:50,105	Signal sent		
20:39:50,130	Element sent		
20:39:51,350		Received signal	
20:39:51,352		Signal sent	
20:39:51,357		Received element	
20:39:51,375		Element sent	
20:39:51,952			Received signal
20:39:51,955			Received element
...	...	...	...

Appendix B

Project Overview

Our implementation is splitted into

- the TeeTime project with the `distributed` branch,
- a test project with an example Pipe-and-Filter architecture and test scripts, and
- the DSL Eclipse plug-in project.

B.1 TeeTime

The `distributed` branch of the TeeTime project is available at `gitlab@build.se.informatik.uni-kiel.de:fec/teetime-distributed.git`.

We extend the maven build, thus it additionally generates `teetime-3.0-SNAPSHOT-distributed.jar` and a `teetime-3.0-SNAPSHOT-distributed-jar-withdependencies.jar`. Thus the standard TeeTime JAR remains small if no distributed features should be used.

To build TeeTime the JDK 8.0 and Maven are required.

B.2 TeeTime-Distributed-Test

Our test project is available at `gitlab@build.se.informatik.uni-kiel.de:fec/teetime-distributed-test.git`.

To build the test project right click the project in Eclipse's package explorer and select `Export`. Then select `Runnable JAR file`. Select `Extract required libraries into generated JAR` and click on `Finish`.

To start the remote system use the following command:

```
java -cp <ProjectJarWithTeeTimeDependency>
    teetime.framework.distributed.RemoteSystem config=<configName>
    config-args=<arg1,...,argN> seed-host=<IP> seed-port=<PORT> host=<IP>
    identifier=<NODE_ID> tcp=<PORT> udp=<PORT> ssl=<PORT>
    ssl-config=<PATH_TO_SSLCONF>
```

Listing B.1. Command to start the remote system

Mandatory arguments are

- ▻ the name of the configuration the remote system should instantiate,
- ▻ the seed host and seed port to join the master,
- ▻ the host to bind the remote system to an IP address it listens on, and
- ▻ the port of the TCP protocol

The arguments

- ▻ `configargs`,
- ▻ `identifier`,
- ▻ `udp`,
- ▻ `ssl`, and
- ▻ `ssl-config`

are optional. When `ssl` is used the `ssl-config` is required. An example ssl-config is available in the `resources` folder. Inside the `ssl.conf` the `key-store` and `trust-store` paths must be modified. The path must be absolute. Akka cannot handle relative paths. The `node.keystore` and `node.truststore` inside the `resources` folder can be used.

The `scripts` folder contains the scripts we used to execute the evaluation tests.

B.3 Teetime-Distributed-Dsl

The DSL plug-in project is available at `gitlab@build.se.informatik.uni-kiel.de:fec/teetime-distributed-dsl.git`. To test the plug-in simply run the sub-project `dsl.DistributedConfig` as an Eclipse application from inside Eclipse.

Bibliography

[Agha 1985] G. A. Agha. *Actors: a model of concurrent computation in distributed systems.* Technical report. DTIC Document, 1985. (Cited on page 11)

[Aldinucci et al. 2012] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Targeting distributed systems in fastflow. In: *European Conference on Parallel Processing*. Springer. 2012, pages 47–56. (Cited on page 65)

[Fowler 2010] M. Fowler. *Domain-specific languages*. Pearson Education, 2010. (Cited on page 12)

[Frank Buschmann 2007] D. C. S. Frank Buschmann Kevlin Henney. Pattern-oriented software architecture, volume 4, a pattern language for distributed computing. In: John Wiley & Sons, Apr. 4, 2007, pages 200–201. (Cited on page 5)

[George F. Coulouris and Blair 2011] T. K. George F. Coulouris Jean Dollimore and G. Blair. Distributed systems. In: Financial Times Prent., Aug. 11, 2011. Chapter 1, page 2. (Cited on page 5)

[Gupta 2012] M. Gupta. *Akka essentials*. Packt Publishing Ltd, 2012. (Cited on page 12)

[Hanmer 2013] R. Hanmer. *Patterns for fault tolerant software*. Wiley Software Patterns Series. Wiley, 2013. (Cited on pages 7–11)

[Hayashibara et al. 2004] N. Hayashibara, X. Defago, R. Yared, and T. Katayama. The/spl phi/accrual failure detector. In: *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE. 2004, pages 66–78. (Cited on pages 16 and 36)

[Otero 2012] C. E. Otero. Software engineering design: theory and practice. In: AUERBACH PUBN, June 11, 2012, pages 116–119. (Cited on page 5)

[Roestenburg et al. 2015] R. Roestenburg, R. Bakker, and R. Williams. *Akka in action*. Manning Publications Co., 2015. (Cited on page 12)

[Silcock and Gościński 1995] J. Silcock and A. Gościński. *Message passing, remote procedure calls and distributed shared memory as communication paradigms for distributed systems*. Deakin University, School of Computing and Mathematics, 1995. (Cited on page 6)

[Sommerville 2012] I. Sommerville. Software engineering. In: Pearson Studium, Mar. 1, 2012. Chapter 6.3.4, pages 200–201. (Cited on page 5)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a generic and concurrency-aware pipes & filters framework (2014). (Cited on pages 1 and 5)

[Wulf et al. 2017] C. Wulf, W. Hasselbring, and J. Ohlemacher. Parallel and generic pipe-and-filter architectures with teetime. In: *International Conference on Software Architecture (ICSA) 2017*. Apr. 2017. URL: `http://eprints.uni-kiel.de/37563/`. (Cited on pages 5 and 65)

[Wulf et al. 2016] C. Wulf, C. C. Wiechmann, and W. Hasselbring. Increasing the throughput of pipe-and-filter architectures by integrating the task farm parallelization pattern. In: *Proceedings of the 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE 2016)*. IEEE, Apr. 2016, pages 13–22. (Cited on pages 1 and 5)

[Zloch 2016] M. Zloch. Development of a domain-specific language for pipe-and-filter configuration builders. Feb. 2016. (Cited on page 37)