Master Thesis

An Online Stream Processor for Timely Dataflow

Author(s):
Wicki, Sebastian

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-010807282

Rights / License:
In Copyright - Non-Commercial Use Permitted
Master’s Thesis Nr. 155
Systems Group, Department of Computer Science, ETH Zurich

An Online Stream Processor for Timely Dataflow

by
Sebastian Wicki

Supervised by
Dr. Desislava Dimitrova
Dr. John Liagouris
Prof. Timothy Roscoe

May 2016 – November 2016
Online processing of data streams is an increasingly common task in many applications. This motivates the use of platforms which not only perform the submission and execution of different computational tasks, but also offer mechanisms for accessing the available data sources.

In this thesis, we present the architecture and implementation of a system for deploying Timely Dataflow applications. Timely Dataflow is an existing framework for writing distributed, data-parallel dataflow programs, with support for iterative and incremental computations. Our system manages the submission and execution of such Timely Dataflow programs in a dynamic cluster of machines. It provides an introspection mechanism which allows computations to inspect the current system state. We extend Timely Dataflow with operators for sharing data streams between different dataflow computations, enabling the composition of separately submitted programs.

We evaluate our system by demonstrating that it can be used to modularize and execute an application which performs statistical processing of reconstructed user-sessions on log traces from a large datacenter.
ACKNOWLEDGMENTS

I would like to thank Prof. Timothy Roscoe, not only for providing me with the opportunity to work on this project, but also for his guidance along the way.

My thanks also goes to my supervisors, Dr. Desislava Dimitrova and Dr. John Liagouris. This work would not have been possible without their great support and invaluable feedback.

Finally, I would like to thank Andrea Lattuada and Zaheer Chothia, as their initial inputs have gained me a better understanding of the particularities of Timely Dataflow.
## CONTENTS

1 **INTRODUCTION**  
1.1 Contributions .................................. 1  
1.2 Outline .......................................... 2  

2 **TIMELY DATAFLOW**  
2.1 Computational Model ............................... 3  
2.1.1 Data-Parallelism ............................... 4  
2.1.2 Progress Tracking ............................... 4  
2.2 Implementation .................................... 5  
2.2.1 Programming Interface ........................... 5  
2.2.2 Worker Threads .................................. 5  
2.2.3 Runtime Graph Representation .................... 6  
2.2.4 Deployment ...................................... 7  

3 **DESIGN**  
3.1 Overall Architecture ............................... 9  
3.2 Queries ........................................... 12  
3.3 Coordinator ....................................... 12  
3.3.1 Submission ...................................... 13  
3.4 Executors ......................................... 14  
3.5 Sharing Data Streams ............................... 15  
3.5.1 Topics .......................................... 16  
3.5.2 Stream Publisher ............................... 17  
3.5.3 Subscriber ...................................... 18  
3.5.4 Comparison with Timely’s Capture & Replay ....... 18  

4 **IMPLEMENTATION**  
4.1 Query Library ...................................... 19  
4.2 Executor .......................................... 20  
4.3 Coordinator ....................................... 20  
4.3.1 Dealing with Asynchronous Tasks ................. 21  
4.3.2 Maintaining Shared Mutable State ............... 22  
4.3.3 Catalog ........................................ 25  
4.3.4 Publishers & Subscribers ......................... 26  
4.4 Shared Communications Layer .................... 30  
4.4.1 Messaging ...................................... 30  
4.4.2 Request & Response Messages .................... 30  
4.4.3 Serialization .................................... 31  

5 **EVALUATION**  
5.1 Query Composition for Sessionization ............... 33  
5.1.1 Computation & Workload ......................... 33  
5.1.2 Dataflow Graph ................................. 34  
5.1.3 Worker-to-Processor Mapping .................... 36  
5.2 Results ............................................. 37  
5.2.1 Overhead in Execution Time ..................... 37
5.2.2 Worker Utilization Factor .................................................. 38
5.2.3 Memory Consumption and Processor Utilization ................. 40
5.2.4 Impact of Worker Placement ........................................... 43
5.3 Discussion ........................................................................ 44

6 RELATED WORK

6.1 Related Work ..................................................................... 45
6.1.1 Dataflow Streaming Engines ........................................ 45
6.1.2 Sharing Dataflow Streams ............................................. 47

7 FUTURE WORK & CONCLUSION

7.1 Future Work ....................................................................... 51
7.1.1 Extracting the Dataflow Graph from Queries ................... 51
7.1.2 Retroactive Tapping of Dataflow Edges ......................... 51
7.1.3 Resource Management .................................................. 53
7.1.4 Backpressure, Buffering and Persistence ....................... 54
7.1.5 Partitioning and Filtering by Subscribers ....................... 54

7.2 Conclusion .......................................................................... 55

BIBLIOGRAPHY ........................................................................ 57
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>System architecture</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Query submission with a single process</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Publisher and subscribers</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Schema of the catalog</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Publish/subscribe sequence diagram</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Dataflow graph for monolithic sessionization</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Dataflow graph for modular sessionization</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Execution times for different number of workers</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Utilization of sessionization workers</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Utilization of subscribing workers</td>
<td>39</td>
</tr>
<tr>
<td>5.6</td>
<td>Peak memory consumption</td>
<td>41</td>
</tr>
<tr>
<td>5.7</td>
<td>Total CPU utilization</td>
<td>42</td>
</tr>
<tr>
<td>5.8</td>
<td>Execution time for different worker placements</td>
<td>43</td>
</tr>
<tr>
<td>7.1</td>
<td>Work deduplication through derived topics</td>
<td>53</td>
</tr>
</tbody>
</table>

LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Terminology of the system</td>
<td>11</td>
</tr>
</tbody>
</table>

LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example program in Timely Dataflow</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Timely’s worker configuration data type</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Example query</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Connection handling at the coordinator</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Handler for blocking subscription requests</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Publish &amp; subscribe interface</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Publishing partitioned topics</td>
<td>27</td>
</tr>
<tr>
<td>4.5</td>
<td>Collection publisher interface</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>Typical use of the subscription handle</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>Request trait</td>
<td>31</td>
</tr>
</tbody>
</table>
INTRODUCTION

Many applications today perform real-time monitoring and analytics on potentially unbounded streams of data. Examples of this include analytics for commercial web applications and social networks, online processing of sensor data, and monitoring and networking in datacenters. In many deployments however, the number of data sources and the amount and complexity of queries will increase over time, and thus the need for management systems and expressive programming models for online stream processing arises.

In this thesis, we investigate the architecture and implementation of a management system for running computations written in the timely dataflow programming model.

Timely dataflow is a programming model for a wide range of data-parallel algorithms. It has first been implemented and introduced in the Naiad system [13]. The timely dataflow model has been successfully used for stream and graph processing, and it is the central building block for higher-level abstractions such as differential dataflow [11]. Timely dataflow supports stateful operators and iterative computations, while achieving high throughput and low-latency responses thanks to a decentralized coordination mechanism for global progress tracking. Our work builds upon the Rust implementation of the timely dataflow model, simply called “Timely Dataflow” [10].

Being an embeddable library, the current runtime of Timely Dataflow only concerns itself with the execution and scheduling of a single processing job. This results in a tendency towards large, monolithic dataflow graphs which perform many different tasks at the same time. While these monolithic computations are highly efficient in terms of resource usage, this comes at a cost with regard to maintainability and extensibility. Attaching an additional stage to a monolithic dataflow graph cannot be done for an online application without restarting the whole computation.

Furthermore, while Timely Dataflow does support the distributed execution of a dataflow computation on a statically sized cluster, launching and deployment of the compiled application binary has to be done manually by the user. This can quickly become inconvenient if many different dataflow computations have to be deployed.

While Timely Dataflow has been successfully used for online streaming applications, we believe that further improvements can be made.

1 We use the capitalized term for the implementation, while the model is referred to in lower-case.
in terms of usability and composability. Specifically, we wish to ease the deployment of multiple concurrent dataflow computations, the management of the compute cluster, and we want to assist with the use of shared data sources.

1.1 CONTRIBUTIONS

The contribution of this thesis is the design and implementation of a system for the deployment of Timely Dataflow applications. It provides the following features:

- Handling the submission and execution of multiple concurrent dataflow programs.
- Dynamic addition and limited removal of machines to the cluster.
- An integrated publish/subscribe system which allows dataflow programs to share their produced streams for consumption by other running computations.
- Enabling system introspection by exposing metadata about the current system state.

1.2 OUTLINE

In chapter 2, we provide a brief overview of the Timely Dataflow library, its conceptual model and its implementation. The system designed and implemented as part of this thesis is introduced and discussed in detail in the chapters 3 & 4. As our system allows the dynamic composition of dataflow programs, we evaluate the overhead of this feature on a realistic workload in chapter 5. A survey of related work is presented in chapter 6, and chapter 7 closes with future work and conclusions.
Timely Dataflow [10] ("Timely") is a Rust library for writing distributed, cyclic dataflow programs. It is a modular implementation of the timely dataflow model introduced by Naiad [13].

2.1 Computational Model

Timely Dataflow is a framework for writing dataflow programs. Dataflow programming is a programming model in which the computation can be represented as a directed graph: The data flows along edges, while the computational logic in the vertices transforms it. Vertices are also called operators in Timely Dataflow, and the edges connecting them are called streams. Notable features of the timely dataflow model in comparison to many other dataflow systems are its support for structured loops, stateful vertices and the option for vertices to be notified if given input rounds of data are completed. [13]

In the timely dataflow model, the messages flowing along edges are annotated with timestamps. Timestamps typically denote input rounds, or within the scope of a structured loop, the iteration count. Timestamps in timely dataflow must define a partial order, allowing the comparison of timestamps for progress tracking purposes: Timely offers user code to be notified when there are no more outstanding messages for a given timestamp at a certain edge. This allows the delivery of messages to happen out of order, while the system is still able to inform operators about the existence of messages in-flight. The progress tracking algorithm of Timely Dataflow is able to provide these guarantees with minimal synchronization, allowing the computation on different machines to advance asynchronously.

Nested subgraphs, called scopes, allow for a hierarchically structured dataflow graph. A subscope can introduce a new own clock domain, where the timestamps of the inner scope are appended to the timestamps of messages entering from outer scopes. When messages leave a subscope, the inner timestamp is stripped. Timely implements scoped timestamps as product types, with the partial order derived from the combined elements.
2.1.1 Data-Parallelism

Timely employs a data-parallel approach for scaling the computation. In the conceptual dataflow graph, the user has to specify a data partitioning scheme on the incident edges of an operator. During execution, the dataflow graph is instantiated on possibly many worker threads, where each worker maintains a local copy of the dataflow graph. As data is pushed along the edges of the conceptual graph, it is sent to the designated worker according to the partitioning scheme. To put it another way, the data emitted by an operator instance on one worker might be pushed to a successor operator hosted on a different worker.

This distribution of work can result in messages being delivered out of order. This motivates Timely to provide the progress tracking mechanism described below, which allows operators to argue about globally outstanding messages.

2.1.2 Progress Tracking

A key feature of the timely dataflow model is its support for fine-grained notification about the advancement of messages in flight, i.e. informing operators about timestamps for which no more messages will arrive. The set of still observable timestamps at a certain input edge at any point in time is derived from the frontier. A frontier $F$ restricts the set of observable future timestamps $S_F$ as follows: $S_F = \{ t_s \mid \exists t_f \in F : t_s \geq t_f \}$ — any future message must have a timestamp that is greater or equal than any timestamp in the frontier. The frontier itself is a subset of non-comparable timestamps (i.e. an antichain) $F = \{ t_1, t_2, \ldots, t_n \in T \mid t_i \parallel t_j \}$, where $T$ is the clock domain of the input stream.

Operators must ask the system to be notified when the frontier advances. This is typically done by asking for notification about the progress of messages with a given timestamp $t$. The notification is served when there are no more outstanding messages with timestamp $t$ or smaller, i.e. there are no more elements in the frontier less or equal than $t$.

In order not to break notification semantics, operators are only allowed to produce messages with a timestamp greater than or equal to $t$ if they hold a capability for timestamp $t$. Operators can obtain capabilities for a certain timestamp either during initialization of the system, or if they receive a message of the given timestamp.
2.2 IMPLEMENTATION

The implementation of Timely Dataflow consists of two separate Rust libraries, the timely library which implements all functionality related to dataflow graph construction and progress tracking, while a second library called timely_communication implements the creation and initialization of worker threads and provides primitives for communication. These communication primitives come in the form of asynchronous unidirectional channels and are used by both workers and operators, though in different ways: Workers use these channels for the exchange of progress tracking messages, while operators use them to push data along the dataflow edges. To ensure proper ordering, data sent to operators hosted on different workers is interleaved with progress tracking messages.

2.2.1 Programming Interface

The Timely Dataflow library provides a domain-specific language for expressing dataflow graphs in Rust. Users create operators (vertices) and connect them to other operators using stream handles (representing edges).

The Timely library contains implementations for common operators such as map or filter, but also for more generic combinators such as unary or binary for implementing custom operators. Many of the provided operators accept user-defined functions to implement parts of their logic.

External input is fed to the dataflow computation using input operators. These provide a handle for client code to push data with assigned timestamps into the dataflow graph. Operator outputs are represented by typed Stream handles, which are used by succeeding operators to connect the stream to their input ports.

2.2.2 Worker Threads

Timely implements a data-parallel approach for its computation. The dataflow graph is instantiated on multiple worker threads, where each worker drives the computation of its local instance. All workers are fully connected and exchange data and progress tracking messages with their peers. Communication between workers is fully asynchronous, consistency is achieved through the progress tracking protocol. Workers can be distributed over multiple machines, meaning that the computation will be hosted by multiple operating system processes. In the current implementation, each process hosts the same number of workers, and processes communicate with each other over the network.
Listing 2.1: A simple single-threaded example program which reads lines from standard input, converts them into integers and filters out odd numbers.

### 2.2.3 Runtime Graph Representation

When the execution on a worker starts, Timely assembles the runtime dataflow graph from the the user-provided dataflow description. The runtime representation consists of a list of operators and a list of edges between them.

When operators are instantiated, they inform the runtime about which connections they have to other operators and provide an implementation of the `Operate` interface, which is used by Timely for progress tracking and operator scheduling.

By implementing this interface, operators describe their inputs and outputs in terms of what capabilities they initially hold and how message timestamps can advance when passing through (the “path summary”). The second part is needed because nested subgraphs are represented to their parent as single operators. By requesting a path summary of every child’s internal structure, the parent has sufficient information to perform progress tracking for its children. The operator interface is also used by parents to inform their children about the initial capabilities and path summaries of their surroundings.

Once construction is complete, the resulting dataflow graph cannot be changed anymore. In addition, Timely requires that every worker hosts exactly the same dataflow graph structure.

During runtime, Timely’s progress tracking will query the operator about its internal progress, requiring the operator to report how many messages were consumed and produced on each input or out-
put edge, and also which internal capabilities it holds on to. Notification about the frontier at the operator’s input edge is also delivered over the Operate interface.

A noteworthy aspect of this implementation is that the operator runtime interface is heavily oriented towards progress tracking, it is completely decoupled from message delivery. As a consequence, this runtime representation of the operator graph does currently not allow for introspection of the operators logic, the type of data being processed or the contents of the message queues of a certain operator.

2.2.3.1 Operator Scheduling and Execution

Each Timely worker is responsible for scheduling the operators in its local dataflow graph instance. Operator scheduling is cooperative and interleaved with progress tracking: Operators typically perform work when progress tracking polls the operator about its internal progress. This means that progress tracking requests generally result in the execution of any user-provided operator logic.

Operators are always polled in a round-robin fashion, as the worker does not know about any of the operators pending messages or other kinds of internal work to be performed.

2.2.3.2 Channel Allocation

As Timely decouples message delivery from progress notification, there is no need for a central message channel registry. The allocation of messaging channels between operators is done bilaterally by the operator themselves, using specialized endpoint allocators provided by timely_communication. Messages are typically sent in batches, where all records within a batch share the same timestamp.

The channel allocator provided by this library is also used by the workers for broadcasting and receiving progress messages. Communication between workers running in different processes communicate over TCP/IP, while workers within the same process use thread-safe FIFO queues to communicate. Serialization for communication over the network is done using a serialization library called Abomination. It trades type safety for high performance serialization and deserialization, and requires the same in-memory data layout at both communication endpoints.

2.2.4 Deployment

Timely allows the computation to be distributed over multiple operating system processes. These processes host all host the same user-provided number of worker threads an can be run on different machines. Starting a computation involves deploying compiled binary
executable on all participating machines and spawning them in parallel. The amount of worker threads needs to be fixed for a given execution of the program, as all the workers are fully connected in order to be able to exchange messages. The addresses of any peer processes are provided programmatically or alternatively read from a text file. Timely provides a function to assemble the configuration from command line arguments. The resulting worker configuration is represented by the `Configuration` data type shown below:

```rust
c enum Configuration {
    /// Use one thread.
    Thread,
    /// Use one process with an indicated number of threads.
    Process(usize),
    /// Expect multiple processes distributed in a cluster
    Cluster(usize, usize, Vec<String>, bool),
}
```

Listing 2.2: Timely’s worker configuration can be one the three options. The parameters in `Cluster` indicates the number of threads per host, the index of the local host, the list of all participating hosts and a boolean flag to enable reporting.
DESIGN

In this chapter we discuss the design of our system for managing timely dataflow computations. It provides two main services which we discuss separately: First, deployment and execution of submitted Timely programs, and secondly, sharing of data streams between running timely dataflow computations.

3.1 OVERALL ARCHITECTURE

Our system consists of multiple components which we briefly describe here: A dataflow program submitted to run as part of the system is called a *query*. Following Timely’s terminology, the execution of a query is done by a group of *worker* threads. Each worker manages the scheduling and progress tracking of the operator in the dataflow graph.

A query is executed by one or more *executors*. Each executor selected to execute a query will fetch and launch the query binary, ultimately spawning the worker threads.

All these components are managed by a central process called the *coordinator*, which stores and exposes the system state in the *catalog*. The catalog includes the list of the available executors, the running queries and what topics they are processing.

In addition to managing query execution, the coordinator also provides services for query composition: Queries can publish the results of their dataflow computation as *topics*, which in turn other queries can subscribe to.
Figure 3.1: Queries (dashed boxes) consist of one or more worker threads (rounded grey boxes) driving the dataflow computation. A query might span over multiple executors, making use of the network for message exchanges between the workers of a query.

The coordinator maintains a connection to every executor and every query process. The state of the whole system is stored in the catalog.
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUERY</td>
<td>User-submitted Timely program running in the system.</td>
</tr>
<tr>
<td>WORKER</td>
<td>Thread belonging to a query, driving the computation of its local dataflow graph instance.</td>
</tr>
<tr>
<td>EXECUTOR</td>
<td>Process designated to host and spawn queries.</td>
</tr>
<tr>
<td>COORDINATOR</td>
<td>Central process managing all the other components.</td>
</tr>
<tr>
<td>CATALOG</td>
<td>Data collection storing and exposing the system state at the coordinator.</td>
</tr>
<tr>
<td>TOPIC</td>
<td>A named and typed representation of an exposed data stream.</td>
</tr>
<tr>
<td>PUBLISHER</td>
<td>Operator for exposing Timely streams by publishing them as a topic.</td>
</tr>
<tr>
<td>SUBSCRIBER</td>
<td>Consumer of a stream emitted by a given topic.</td>
</tr>
</tbody>
</table>

Table 3.1: Terminology of the system.
3.2 QUERIES

A query is a Timely program managed and executed by our system. Like standalone Timely Dataflow programs, queries are written in Rust and linked against the Timely Dataflow library. The dataflow graph is constructed by connecting Timely’s operators (vertices) to stream objects (edges).

In standalone Timely applications, the user has to manually provide a configuration for the worker threads. In our system, this information is partially generated by the system, based on a user-provided template. Thus, we require that a query registers its computational logic with our `timely_query::execute` function, instead of calling Timely’s initialization function directly.

This means that in order for a Timely Dataflow program to become a runnable query in our the system, it needs to link against our `timely_query` library. This library not only performs the initialization of the Timely runtime, it also provides additional functionality to interact with the coordinator, which we describe below in section 3.5 Sharing Data Streams.

Other than this, we do not want to impose any restrictions on what a query program can do, it is free to execute arbitrary code.

```rust
extern crate timely;
extern crate timely_query;

use timely::dataflow::Scope;
use timely::dataflow::operators::{Filter, Inspect, ToStream};

fn main() {
    timely_query::execute(|root, coordinator| {
        rootscoped::<u32, _, _>({scope} {
            (0..100).to_stream(scope)
                .inspect(|x| println!("hello {x}"));
        });
    }).unwrap();
}
```

*Listing 3.1: Example query which prints out a stream of integers.*

3.3 COORDINATOR

The coordinator is the central process of the system, managing all other components. The coordinator provides an interface for users to submit new queries and inspect the current state of the system. In order to receive commands and report their internal state, every executor and query process maintains a network connection to the coordinator.
Bookkeeping of the system is done by the coordinator in the catalog. The catalog is a datastructure which contains the metadata about the available executors, the running queries and their workers. This data is exposed to queries through so called collection topics, which we will describe below. This allows queries to introspect the system state using Timely operators.

3.3.1 Submission

A query is submitted to the coordinator as a binary executable. Compilation is done externally to allow the use of third-party libraries. A submission consists of the location of the query binary, as well as the runtime configuration for the workers. The runtime configuration specifies the amount and distribution of the worker threads which will drive the query.

Optionally, a human-readable description of the query, as well as the command-line arguments to be passed to the executable can be provided.

When handling a new query submission request, the coordinator will assign a unique identifier to the incoming query, and then select a matching number of executors for the query to be spawned on. The selection of executors is based on the runtime configuration provided by the submission request.

The coordinator plays an important role when spawning new queries. After issuing query spawn requests to the executors, it waits for all query processes to register themselves at the coordinator before they begin their computation. Only then the query is considered active and the submission request is reported to be completed.

Figure 3.2: Submission of a query on a single executor. Only when the spawned query announces itself at the coordinator is it considered running.
3.4 Executors

The spawning and direct supervision of query processes is not done by the coordinator itself, but is offloaded to designated processes called executors. This choice allows for more flexibility regarding query execution and the management of available resources. Since executors can be dynamically added to the system, and with some limitations also be removed again, they provide a way for adding new machines to the cluster running our system. The catalog maintains the pool of available executors which are participating in the system.

As a timely program can span over multiple machines, a query might span over multiple executors, allowing it to run in a cluster. The placement of a query on the available executors is performed by the coordinator, based on the users request. We currently implement a naive approach to query placement, where the executors are chosen randomly if the user does not manually specify a placement. A more sophisticated scheduling, which for example could include load balancing, has to be investigated.

Another feature of executors is that they define how queries and their worker threads are supposed to be executed. When an executor joins the system by introducing itself to the coordinator, the executor also informs the coordinator about the execution format it supports. In the current implementation, executors only support the execution of queries in the form of native operating system executables.

When spawning such an executable, the newly spawned query process performs the worker initialization on behalf of the executor. The executor supplies the newly created process with the information it needs in order to participate in the system. This includes the query’s own identifier, the address of the coordinator, the addresses of any peer processes belonging to the same query, and the number of worker threads to spawn inside this process.

Executors are also responsible for fetching any query binaries they are supposed to spawn. When submitting a new query, the submitter has to provide the coordinator with the location of the binary. This location is forwarded to participating executors, which will use it to load the query binary.

Since executors serve as a provider for computational resources such as machines, there is typically a one-to-one mapping between executors and the machines they are running on. This however is not a requirement of the system, the deployment of executors is left to the user. Similarly, the user is free to spawn multiple processes belonging to the same query on the same executor if they wish to do so.

Since operating system processes can outlive their parent process, executors can be removed from the system while the queries spawned by the executor are still running. In this case, the coordinator removes...
the terminating executor from the pool, disallowing any future queries to be hosted on the removed executor.

**Future Considerations** The current implementation of executors which run queries in their own separate process is not the only possible implementation choice. Alternative implementations could for example host multiple queries within the same process, by dynamically loading the query’s code into an already running process and run it on a preemptive operating system thread. This would allow queries the interchange of objects without the need for data serialization. Given the implementation of Timely’s operator scheduling, cooperative scheduling of multiple worker threads within a single operating system thread could also be implemented for certain queries where its input is managed by the system.

### 3.5 Sharing Data Streams

Dataflow programs typically work on streams from external sources. As the same data source might be of interest for different dataflow computations, it seems appropriate to manage data streams in our system as well. Furthermore, input streams might not only come from external sources: A dataflow computation might produce an intermediate or final output stream which could be of interest for other queries. These assumptions motivate us to extend our system with a mechanism to allow queries to expose their data streams for consumption by other queries.

We implement this in the form of a topic-based publish/subscribe system: Using the `publish` operator, a query can expose one of its streams (an edge in the dataflow graph) to other queries, which in turn then `subscribe` to it. The list of all published topics is stored in the catalog.

The reason for choosing a topic-based publish/subscribe system over a content- or type-based one is dictated by the fact that Rust allows for little type introspection. In addition, having users explicitly publish and name exposed streams helps to differentiate between streams with the same datatype but different semantics.

For providing access to external inputs, we require that a designated query translates these external inputs into Timely streams and publishes the result as a topic. We prefer this manual approach over providing our own adapters, as there already exists a rich collection of Timely-aware code which performs this translation of external input to Timely streams.
3.5.1 Topics

A topic has the following properties, which are all stored in the catalog and can be accessed by other queries.

**Identifier** A unique identifier for the topic instance.

**Name** When publishing a stream, the publisher has to assign a name to it. Queries use this name to refer to topics they want to subscribe to. There might be only one topic with a certain name at a time, however names can be reused if topics are unpublished.

**Schema** A descriptor of the data type of the stream published in this topic. Timely’s streams are typed channels, therefore so are topics.

**Address** An address to which the subscribers connect in order to received the contents of the published stream.

While every publication and subscription request is disclosed to the coordinator and the catalog contains a list of all existing topics, the actual exchange of data happens directly between queries. When a query subscribes to a topic, it receives the address of the topic’s publisher from the coordinator and directly connects to it.

*Figure 3.3:* When publishing some data stream, the producing query can decide to merge the different stream partitions.
3.5.2 Stream Publisher

A stream publisher is a Timely operator which exposes a Timely stream as a topic. When creating it, the query author has to assign a name for the topic under which the input stream will be published. As with all other Timely operators, the publisher operator is instantiated on all worker threads. However, the user can choose whether all worker streams are merged into a single topic before publishing, or if each worker publishes its own topic. The latter option implicitly exposes the data sharding strategy of the publishing query, allowing the workers of the subscribing query to exploit this partition scheme.

3.5.2.1 Collection Publisher

Our publishers are not buffered, meaning subscribers will not receive any data which was produced before they subscribed to a certain topic. This is the same as in many other publish/subscribe systems, where synchronization between publishers and subscribers is decoupled as well. [5]

However, in streams where the contents of the stream describes changes of a certain state, it is essential for stream consumers to know the state of the source at the beginning of the stream in order to make sense out of it. The motivating example here is the catalog, which is supposed to expose the current system state to new participants.

For this reason, we introduce a second kind of publisher which publishes collections instead of streams. A collection in our model is a typed, unordered multiset maintained by the publisher, possibly representing state it would like to share with subscribers.

Upon creation, a collection publisher contains an empty collection. When the publishing query mutates the collection by adding or removing elements, these changes are propagated to the subscribers. When new subscribers connect to this publisher, the publisher will provide them with a list of all currently contained elements. This way, all subscribers eventually maintain the same view of the data collection.

From the subscribers point of view, a topic published by a collection publisher is not inherently different from a normal topic: After subscribing, it will observe a continuous stream of data. The difference is in the data type of the stream, it will be of tuples of the format (Data, δ), where Data is an element that can be stored in the multiset and δ is a signed integer denoting the amount of elements that have been added or removed from the set. This format is compatible with the notion of collections in Differential Dataflow [11], allowing subscribing queries to further process the collection in a convenient manner. Note however that our collections are not versioned, we do not append timestamps to the update messages.
3.5.3 Subscriber

In order to subscribe to a topic, the subscribing query has to provide the name of the topic that it is interested in. This involves a name lookup which can optionally be blocking: If a requested topic does not yet exist, the coordinator will add the subscriber to a wait list. Once a corresponding topic is published under that name, the coordinator will inform the subscriber about this topic.

The subscribing query is free to subscribe to multiple topics. It is the query’s responsibility to merge and feed the data streams into the dataflow graph.

3.5.4 Comparison with Timely’s Capture & Replay

The Timely library provides special capture and replay operators which also serve the purpose of sharing data between queries. With the capture operator, all timestamp, data and event records are collected in one dataflow computation, and can then be replayed in another one.

The capture/replay operators will record and replay the whole stream from beginning to end, requiring the capture operator to either buffer its incoming data or wait for the replaying query to connect to it. In our system, we opted for a more dynamic approach, where producing queries are allowed to discard data if there are no consumers.

It is the responsibility of the user to provide a communication channel between capture and replay operators. Adapters exist for Rust’s thread-safe FIFO queues, single-threaded queues, file handles, and sockets.
IMPLEMENTATION

In this chapter, we discuss the implementation of the individual components presented in the previous chapter. We also present some shared implementation features, such as the common request and response protocol used for the communication between the different components and the coordinator.

4.1 QUERY LIBRARY

As discussed in the previous chapter, queries link against a small wrapper library which allows submitted Timely programs to participate in our system. This library performs the initialization of the dataflow computation, announces its readiness at the coordinator and provides methods for publishing and subscribing to topics.

For the query to announce itself in the system, the query author must eventually call the `timely_query::execute` function. This function mirrors Timely’s own `execute` function in that it performs the execution of the computation. In contrast to Timely however, our version does not provide any means to specify the runtime configuration of the execution. This information is specified by the user at submission time and directly provided to the query library by its hosting executor. The current implementation of the executor passes this information down to the query library in environment variables.

Using this information, the query process connects to the coordinator and registers itself by providing its identifier and the group of workers it will host. Once all worker groups have successfully registered themselves at the coordinator, the coordinator replies with a randomized token which is used to associate the registered processes with the query.

The initialization of the Timely worker threads and the allocation of the communication channels among them is by the library done using `timely_communication` the same way as it would be done in a standalone Timely program. For this reason, our query library needs to translate the information provided by the executor into Timely’s own format.

Our query library provides an interface for worker threads to publish or subscribe to topics. For this reason, every worker thread gets a handle for the connection to the coordinator. Access to the publish & subscribe system is provided through a set of remote procedure call stubs which are described separately in section 4.3.4.
4.2 Executor

The current implementation of the executor is relatively simple, it is a process that spawns child processes on behalf of the coordinator, enabling the execution of new queries on remote machines.

In order to add a new machine to the cluster, the user has to deploy the executor binary on the new host, specify the address of the coordinator and then launch the executor binary. The executor will register itself at the coordinator, which in turn assigns a unique identifier to the connecting executor. Once connected to the coordinator, the executor listens for incoming spawn requests. If such a request arrives, it fetches the binary from the specified source and launches it as an operating system process. During submission, the user can specifying command line arguments which are provided to the query binary here as well.

The executor has to inform the spawned binary about its assigned query identifier, the address of the coordinator and the address of any peer processes. In the current implementation this is simply done by the executor writing this information into the environment variables of the spawned query.

The executor currently uses the process spawning functions provided in Rust’s standard library, which at present offer all features we need, and are available on all supported operating systems. As we will discuss in later chapters, more sophisticated, platform-specific process control and supervision mechanisms could be added in the future.

When submitting a new query, the submission must contain an URL to the binary. The URL scheme determines how the binary is fetched by the executor. In the current implementation, this can either be a filesystem path directly accessible by the executor process (e.g. over a shared network folder), or the binary can be provided over a raw TCP stream. More advanced download schemes can be easily added in the future.

4.3 Coordinator

As the central component of our system, the coordinator performs a rich set of tasks. These involve handling submission requests to spawn new queries, but also handling publication and subscription requests from existing queries. It is the central authority on the current system state, i.e. it maintains a list of all running queries and executors, but it also tracks all existing publications and subscriptions. It exposes this information through the catalog.

Because most components interact with the coordinator, its internal implementation is not without challenges, particularly when dealing with multiple concurrent events.
Before we discuss the details of its implementation, we provide a brief overview of the internal architecture of the coordinator: To other components, the coordinator exposes a request/response-based interface on a predefined networking port. It listens for incoming connections on that port and then waits for the connected clients to send their requests. Once a request is received and decoded, it is forwarded to a central request handler. This request handler contains the central logic of the coordinator, keeps track of unfulfilled requests, and informs the catalog to announce changes in the system state.

4.3.1 Dealing with Asynchronous Tasks

Our initial prototype of the coordinator used a multi-threaded approach for dealing with multiple clients at the same time, and used message passing between threads to avoid directly exposing shared state. However, many requests handled by the coordinator require it to wait for external events, and thus a form of cooperative task management is needed.

We initially used a continuation-passing style for splitting blocking requests into non-blocking subtasks. However, due to Rust’s memory ownership model, we found that this approach often resulted in manual stack ripping [1], which made code both hard to read and inconvenient to write.

This motivated us to re-design the coordinator around a central task dispatch loop, which allows us to multiplex many asynchronous tasks within the same thread. We decided to adapt an external library called future-rs [16] for this purpose. It provides an expressive interface for dealing with asynchronous tasks based on the concept of futures. Futures (sometimes called promises, eventuals or deferred objects) are proxy objects for absent values, which eventually will be provided through some asynchronous event. The future-rs library provides combinators for chaining futures together or waiting on multiple futures at the same time. Actions on completed futures are expressed as closures. This approach does not completely eliminate stack ripping, however the fact that the completion handlers are chained together allows for much more maintainable and sequentially looking code. Listing 4.2 shows how futures can be used when executing potentially blocking requests.

A unique aspect of the implementation of future-rs is that its futures need be polled in order to make progress, instead of proactively being called by event sources. When a future is polled, it checks if any pending events have occurred. If so, it dispatches any pending completion handlers. It is the users responsibility to ensure that futures are being polled. This is typically done by registering the future in an event loop, which will repeatedly wait for events to occur before polling its registered futures.
4.3.1.1 Task Dispatcher

The future-rs library itself provides a set of primitives to create and combine futures. It also provides abstractions for expressing which events a future is waiting for, and expressing the occurrence of events. It does however not provide an actual implementation of an event loop. Our implementation of the coordinator thus provides a simple task dispatcher whose sole purpose it is to wait for events to occur and poll registered futures accordingly.

To avoid having to pass around a handle to the dispatcher, we store its list of pending futures in thread local storage. The public interface consists of two free-standing functions, `async::spawn` for registering futures which are to be completed, and `async::finish` which initializes the dispatcher loop and then waits for its termination. `async::finish` accepts an initial future which acts as the root task. The motivation for the names in this interface comes from the fact that model each concurrently running task as a chain of futures, basically treating them as coroutines.

Listing 4.1 shows an example of this. Our networking layer models the server socket as a future which yields a stream of incoming connections. Each connection itself is then maintains a queue for incoming requests, which is also modeled as a future, allowing the user to specify actions to be taken once a request is received.

4.3.2 Maintaining Shared Mutable State

The nature of the coordinator requires it to mutate its internal state on behalf of the connected clients. This means that we have to provide each connection handler with mutable access to the central request handler. In addition, we also want to keep track of the requests submitted by each connection, so we can clean up any associated global state if a connection disappears.

For this reason, we provide a wrapper type for accessing the methods of the request handler. For each connection, this proxy maintains a simple list of identifiers for objects that are conceptually owned by the connection, ensuring that the state is removed from the catalog once the connection drops. Examples of this are the removal of disconnected executors from the catalog, or the depublication of topics owned by crashed queries. The proxy object internally uses Rust’s `RefCell<T>` type to dynamically ensure the strict language rules on shared mutability.
// a future yielding a stream of accepted connections
let listener = network.listen(9189);

// define the action to be executed for each incoming connection
let server = listener.for_each([connection] {
  ...
  // this action is invoked for each incoming request
  let client = requests.for_each(move |request| {
    // decode and dispatch request
    match request.name() {
      "Subscribe" => {
        let (req, resp) = req.decode::<Subscribe>();
        // forward the request to the request handler,
        // which will return a future (see Listing 4.2)
        let subscribe = request_handler
          .subscribe(req)
          .then(|result| Ok(resp.respond(result)));
        // complete this task asynchronously
        async::spawn(subscribe);
      }
      ...
    }
  })
    .map_err(|err| {
      // futures can also yield error values
      // which are handled separately
      error!("failed to dispatch request: {?:}", err)
    });

  // handle client asynchronously
  Ok(async::spawn(client));
});

// drive the event loop to completion
async::finish(server);

Listing 4.1: Accepting and dispatching incoming connections. The actions
specified in for_each and map_err are invoked by the task dispatcher when
the corresponding events (such as new incoming connections, new incoming
requests or errors during request handling) occur.
struct RequestHandler {
    catalog: Catalog,
    // list of pending lookups
    lookups: HashMap<String, Vec<Complete<Topic, SubscribeError>>>
}

impl RequestHandler {
    pub fn subscribe(&mut self, request: Subscribe) -> Box<Future<Item=Topic, Error=SubscribeError>> {
        // extract arguments from the request
        let query = request.query.id;
        let name = request.name;
        // first, check if the topic actually exists
        if let Some(topic) = self.catalog.lookup(&name) {
            // if so, insert a new subscription into the catalog
            self.catalog.subscribe(query, topic.id);
            return futures::done(topic).boxed();
        } else if request.blocking {
            // allocate an unresolved future and a completion handle
            let (lookup, result) = promise();
            // to be executed once the topic exists
            let handler = self.handle();
            let result = result.and_then(move |topic| {
                handler.catalog.subscribe(query, topic.id);
                Ok(topic)
            });

            // register the completion handle for pending lookups
            self.lookups.entry(name).or_insert(vec![]).push(lookup);

            return Box::new(result);
        } else {
            // non-blocking subscription request, fail immediately
            let err = SubscribeError::TopicNotFound;
            return futures::failed(err).boxed();
        }
    }
}

Listing 4.2: Example how a future is created for the subscription request. Depending on whether it can be served immediately or not, different kinds of futures are returned. Bookkeeping for still pending requests is done with completion handles. Upon publication of a requested topic, any pending subscriptions will be completed by calling lookup.complete().
4.3.3 Catalog

The purpose of the catalog is to maintain and expose the current state of the system. It tracks the addition and removal of executors, queries, topics, publication and subscriptions and exposes them in collection topics according to the schema shown in Figure 4.1.

Figure 4.1: Schema of the catalog. All five data types are published as a collection topic, allowing users to follow state changes in the system.

The catalog itself only exposes this data. It is mutated and queried by the request handler in order to fulfill the submitted requests.
4.3.4 Publishers & Subscribers

The query library exposes the publish and subscribe functionality. The following API is used for publishing and subscribing to Timely streams.

```rust
pub struct Coordinator {
    /* hidden handle to request queue */
}

impl Coordinator {
    pub fn publish<S, D>(&self, name: &str, stream: &Stream<S, D>,
        part: Partition) -> Result<Stream<S, D>, PublicationError>
        where D: Data + NonStatic,
            S: Scope,
            S::Timestamp: NonStatic;

    pub fn subscribe<T, D>(&self, name: &str, cap: Capability<T>)
        -> Result<TimelySubscription<T, D>, SubscriptionError>
        where T: Timestamp + NonStatic,
            D: Data + NonStatic;
}
```

Listing 4.3: The interface for publishing and subscribing Timely streams. The NonStatic bounds on the data and timestamp parameters are required for safe serialization described in 4.4.3.

4.3.4.1 Publisher

The publish function takes a direct reference to a Timely Stream<<, >>. These stream handles, which are only available during the dataflow graph construction phase, allow us to instantiate our own publish operator on the stream. The instantiated publish operator will push its incoming data as well as its current frontier to any subscribers.

Once the operator is inserted into the dataflow graph, the publish stub issues a registration request to the coordinator, which will result in the creation of a topic for other queries to subscribe to.

The partitioning argument on the API specifies whether all streams shall be merged into a single topic, or if every worker publishes its own stream. Merging of the stream is done using Timely’s partitioning functions. In the second case, the identifier of the publishing worker is appended to the name of the topic.
timely_query::execute([root, coord] { 
    rootscoped::<u64, _, _>({scope} { 
        let i = scope.index();
        let numbers = (i*100..(i+1)*100).to_stream(scope);

        // results in 'n' topics:
        // "numbers.0", "numbers.1", ..., "numbers.n"
        coord.publish("numbers", &numbers, Partition::PerWorker) 
            .expect("failed to publish topic");

        // filtering performed by each worker in parallel
        let primes = numbers.filter(|x| x.is_prime());

        // results in a single merged topic called "primes",
        // published by worker with index 0
        coord.publish("primes", &primes, Partition::Merge) 
            .expect("failed to publish topic");
    });
})

Listing 4.4: Publishing partitioned topics.

collection publisher  In addition to the stream publisher, we also provide the so called collection publisher. It has a similar interface for publishing, but it requires the type of the incoming stream to deliver (D, i32) tuples, where the integer denotes the amount of elements that have been added or removed from the collection.

```rust
impl Coordinator {
    pub fn publish_collection<S, D>(&self, name: &str,
        stream: &Stream<S, (D, i32)>, partition: Partition) 
        -> Result<Stream<S, (D, i32)>), PublicationError>
    where D: Data + NonStatic,
        S: Scope,
        S::Timestamp: NonStatic;
}
```

Internally, the publisher maintains a copy of this collection. This is required for new subscribers, which must be informed about the contents of the collection when they connect.

implementation details  Both publishers share the same underlying logic which accepts incoming connections from subscribers and provides notifications for disconnected clients.

Internally, this logic also make use of the future-rs library, since futures have also been integrated in our networking layer. Inside the coordinator the polling is done by the task dispatcher. For the futures inside our publish and subscribe operators however we make use of
the fact that Timely itself also uses polling-based scheduling. Every time the publish operator is scheduled by Timely, it can poll the networking layer to be informed about newly accepted or disconnected subscribers.

Note that even though each subscription request is disclosed to the coordinator by subscribing queries, publishers are not informed by the coordinator about new subscribers. Publishers see new subscribers only once they connect to the publisher’s network socket. The information stored in the catalog is only exposed for inspection by the user. Figure 4.2 shows hows the sequence for publishing a new topic.

![Sequence Diagram](image)

*Figure 4.2: Sequence diagram for the publish/subscribe procedure. Any data produced before the subscriber connects might be lost.*

### 4.3.4.2 Subscriber

In contrast to the publisher, the current implementation of the subscribe function does not directly instantiate a Timely operator, but returns a subscription handle which is used to read data from the topic.

There are two reasons for this choice: First, Timely requires that an operator is instantiated on every worker, such that all instances of the dataflow graph are identical. However, the amount of topics a query would like to subscribe to, and the amount of workers it allocates might not match. Thus, the API quickly becomes complicated, as we would need to provide a way for the query author to map topics to operator instances.

Second, Timely’s operator contract requires that an operator instance announces the internal initial capabilities it holds, and, more importantly, the initial capabilities of its peers. To support this would require some synchronization between the subscribers, as they have to exchange the initial frontier they observe at their topic.
Instead, we decided to base our subscription progress tracking on Timely’s new capability handles. The recently added *unordered input operator* allows a query to feed a computation from data with unordered timestamps. This operator exposes a root capability for the earliest possible timestamp, allowing the user to derive capabilities for newer timestamps from old ones. The frontier is advanced by dropping the capabilities for timestamps for which no more data is pending.

Based on the progress tracking information delivered by the publisher, the subscription handle will automatically derive new capabilities for incoming data and drop old ones if the frontier advances. The user however has to provide the initial root capability for the input.

```rust
Listing 4.6: Typical use of the subscription handle. This query subscribes to a single topic of strings, with u64 being the type of the timestamps.

timely_query::execute(|root, coord| {
    let (mut input, cap) = root.scoped::<u64, _, _>({scope} {
        let ((input, cap), stream) = scope.new_unordered_input();
        // stream/operators(..)
        (input, cap)
    });

    let topic = coord.subscribe::<_, String>("example", cap)
        .expect("failed to subscribe");

    for (time, data) in topic {
        let session = input.session(time);
        session.give_content(&mut Content::Typed(data));
        root.step();
    }
})
```

In queries which use multiple workers, the query author can programmatically encode which workers subscribe to which topics, as the code outside the graph generation is allowed to conditionally decide if and how it wants to call the subscribe function. This functionality is required as there is not always a one-to-one mapping between published topics and workers at the subscribing query, and expressing the assignment in arbitrary Rust code gives the most flexibility to the query author.

Because the root capability can be cloned, this interface also allows the user to interleave the data from a topic with other data, for example by merging multiple topics at the input. Our subscription handle implements the *futures-rs* stream interface, allowing the client code to wait for data from multiple subscriptions at once.
4.4 SHARED COMMUNICATIONS LAYER

For the remainder of this chapter we discuss the networking layer of our system, which is used by all components to implement their services.

4.4.1 Messaging

Our system consists of potentially many distributed processes. These processes are running concurrently, are dynamically added and removed from the system and are communicating with each other. In order to deal with the inherent complexity of such a system, we adopted an actor-model like approach for our networking implementation.

Actors in our system include processes like the coordinator, the executors and the query processes. But we also treat the publish or subscribe operators as actors, as they directly interacting with our system.

To support this kind of model, our networking layer also works in terms of asynchronous messages. In contrast to Timely’s own networking layer which also provides a similar abstraction, we cannot assume a fixed number of components. In order to establish a connection between two components, one of them has to take on the role of the server, while the other one acts as a client. Two queues are allocated per connection on each side: one for incoming messages and one for outgoing messages. This enables messages to be sent asynchronously in both directions. Network failures are signaled in the queue for incoming messages. Currently, Rust’s standard library TCP sockets are used as the underlying transport mechanism, however the system could easily be extended to support alternative transport layers as well.

4.4.2 Request & Response Messages

While the abstraction of single messages is sufficient for implementing the mostly unidirectional messaging paradigm of the publish-subscribe implementation, most other communication in the system follows a request-response pattern, e.g. any communication with the coordinator.

For this reason, we implemented a request-response multiplexer on top of the plain message channels. This multiplexer allows both sides to have multiple requests in flight while waiting for the corresponding responses, which can be delivered out of order.

In order to differentiate between different kind of requests and also ensure a well-typed response format, request payloads have to implement the Request trait. The associated name is used for decoding, while the associated Success and Error types specify what kind of pay-
loads are valid for the response. This allows the response for a given request $R$ to be represented by Rust’s $\text{Result}<R::\text{Success}, R::\text{Error}>$ type.

```
pub trait Request: Abomonation + Any + Clone + NonStatic {
    type Success: Abomonation + Any + Clone + NonStatic;
    type Error: Abomonation + Any + Clone + NonStatic;

    fn name() -> &'static str;
}
```

Listing 4.7: In order for a type to be used as a request message, it needs to implement the `Request` trait. The name allows request handlers to differentiate between different types of requests, while the associated types forces them to issue well-formed responses. The trait bounds are explained in section 4.4.3.

### 4.4.3 Serialization

For both, the request/response messages as well as messages used in the publish/subscribe subsystem, we need to serialize the payloads in order to send them over the network.

#### 4.4.3.1 Message Buffers

Incoming and outgoing network data is stored in reference counted message buffers. This buffer supports multi-part messages, meaning a message contains more than one serialized object. This is needed for the request/response multiplexer, which needs to partially decode a message in order to determine to which request it belongs to. It is also used in the publisher and subscribers, which allows them to encode and decode the timestamp and the data separately.

Reference counting is an optimization used by publishers, which allows them to serialize the data only once and share the buffer with all subscribers. Every outgoing queue to the subscriber only contains a reference to the buffer. Once the message is written to the subscriber’s socket, the reference count is decreased and the message is deallocated once all subscribers have consumed it.

#### 4.4.3.2 Safe Serialization with Abomonation

Timely itself provides a uses a high-performance serialization library called Abomonation, implying that all data types sent to a publish operator will be serializable this way. This makes Abomonation a natural choice as the serialization format for the messages sent from publishers to subscribers.

However, Abomonation is neither memory- nor type-safe. There are no safeguards against deserializing data into an incompatible
type, which will result in undefined behavior. In Timely, such errors are typically avoided since all worker execute the exact same program code and the streams between workers are statically typed. In our system however, a publisher might accidentally use a different version of a library type than the subscriber. Because of this, we use our own lightweight wrapper around Abomonation. In addition to the raw serialized bytes, we annotate the buffer with the TypeId of the data type. Rust’s TypeId provides an opaque, globally unique identifier for a given Rust type and its representation. Thus, we can check if the expected and provided type identifier match before trying to deserialize a message.

In order for a type to be serialized by our library, it needs to fulfill the following type bounds: Abomonation + Any + Clone + NonStatic.

The Any trait is required to retrieve the type’s identifier. The Clone trait is used to put deserialized types into the incoming messages queue and NonStatic is an auto-trait used to disallow the creation of eternally valid pointers into temporary buffers. We also take type alignment rules into consideration when serializing into unaligned buffers.

**Alternative serialization formats** We currently use our safe Abomonation wrapper not only for data in publish/subscribe system, but also for serializing and deserializing requests and response messages. This implies that currently all participating components have to be compiled for the same processor architecture and operating system.

Because of this, our message buffer interface has been designed to use alternative serialization formats in the future. Besides supporting heterogeneous clusters, alternative formats might also be useful in the publish/subscribe system: Schema-based serialization formats would allow subscribers to decode published data even without static knowledge of its data type.
EVALUATION

For our experimental evaluation we split a monolithic dataflow computation into a modular set of queries. The modular queries share their results in a topic, allowing subscribers to further process the data. Because queries can be dynamically added to the system, there is no need to interrupt the running computation if additional analysis stages have to be added.

This increased flexibility comes however at a cost. As data has to be published in topics in order to be accessed by consuming queries, we expect there to be some overhead in both memory consumption and processing latency. The goal of this chapter is to measure and analyze the overhead of query composition on a realistic workload.

Experimental setup All experiments are executed on a dual-socket Intel Xeon E5-2650 machine (2.0 GHz, 8 cores per socket, 2 threads per core) with 64 GB RAM. All components and queries in our system are compiled with a Rust 1.14 nightly build (2016-10-20) and executed on Debian 7.8. The input data is read from a SAN storage attached via iSCSI on a private 10 Gbit Ethernet network.

5.1 QUERY COMPOSITION FOR SESSIONIZATION

The dataflow computation in this experiment is based on sessionization, i.e. the reconstruction of user sessions from a trace of log events.

5.1.1 Computation & Workload

The workload in this experiment is an hour-long trace of log events, collected in a large datacenter owned by an provider for the travel industry. The messages in our trace were accumulated concurrently by 42 log servers, which received their log events from 1263 different streams. Events are originating from a distributed middleware which acts as a message broker. It records metadata about the received messages, such as timestamp, session identifier, and transaction identifier, and forwards them to the log sever.

The computation we use on this workload is called sessionization. Sessionization is the reconstruction of user sessions, i.e. grouping and then processing all messages belonging to the same session. For this purpose, the computation waits for the arrival of late messages before it can consider the session closed. In our experiment we use a fixed inactivity limit of 5 seconds, and re-order messages within an event
time window of 10 seconds. This is enough to incorporate more than 99.99% of all messages.

While sessionization has been designed for real-time processing on a live stream of events, we will load the workload trace from disk for our experiments. This not only simplifies the experimental design and ensures reproducibility, it also allows us to explore the behavior of the system if there are only a few worker threads.

5.1.2 Dataflow Graph

Input is read from disk and fed into the dataflow graph in parallel, each worker is responsible for a partition of the log servers. The read messages are sent into the sessionization stage, where all messages belonging to the same session are grouped together.

After the sessionization, the computation performs a set of statistics on the reconstructed sessions. The stages are described below, the resulting dataflow graph is shown in Figure 5.1.

Messages are shuffled according to their session identifier for the sessionization stage, and are processed in parallel until statistics are collected for the output, which is emitted at a single worker.

**INPUT** Parses the input files and feeds them into the sessionization stage.

**SESSIONIZATION** Distributes and then groups messages according to their session identifier. Emits groups of messages belonging to the same session after a period of inactivity.

**MESSAGES PER SESSION** Simply counts and emits the number of messages belonging to a session.

**DURATION OF SESSION** Calculates the timespan between the first and last message of a session.

**PARSE TRANSACTION TREES** Parses the transaction identifiers for tree extraction.

**TRANSACTION TREE DEPTH** Calculates the depth of the transaction trees.

**TOP K TRANSACTION TREE SIGNATURES** Calculates signatures for the transaction trees and reports the ten top most common tree signatures.

**TOP K COMMUNICATION PAIRS** Infers the ten most common pairs of services who are communicating with each other.

**OUTPUT** Collects emitted data in a histogram.
Figure 5.1: The dataflow graph for the original, monolithic sessionization query.

Figure 5.2: Structure of the modular sessionization. Two topics are introduced connect the tree of queries.
5.1.2.1 *Splitting the Dataflow Graph*

For this evaluation we split the computation such that each stage represents its own sub-query. In order to achieve this, we introduce two topics: The topic `sessionize` contains the result of the sessionization stage, i.e. the groups of messages belonging to a session. The second topic, named `transactions`, is introduced for the transaction-oriented statistics. It emits a stream containing all the transactions per session. The resulting modular dataflow graph is shown in Figure 5.2, it consists of seven queries, each one running in its own operating system process.

**TOPIC PARTITIONING**  We maintain a simple one-to-one mapping between the number of topics and number of worker threads in both the publishers and the subscribers. Merging the stream partitions for publication and redistributing the data at each subscriber would just result in unnecessary work.

5.1.3 *Worker-to-Processor Mapping*

We use the same amount of workers for each query to simplify topic partitioning. But because the modular version naturally consists of more than one query, the execution of the modular version will spawn more operating system threads than the monolithic version.

For this reason we limit the available CPU cores for the different executions, ensuring the number of workers per query matches the number of available CPU cores. Because our machine does have two processor sockets, each with their own memory bank, the question arises how performance is impacted by different processor placement policies. We explore this in section 5.2.4.

If not stated otherwise, we prefer using as many physical cores as possible before falling back to hyper-threads, and we prefer filling the first socket before allocating threads on the second socket. The memory allocation policy is set up to always allocate on the local node.

Note that we do not pin individual worker threads to individual cores, we just disallow access to excess cores. The operating system is free to load-balance threads between the available cores.
5.2 RESULTS

5.2.1 Overhead in Execution Time

We report the overall execution time of both the monolithic and the modular query for different amounts of worker threads in Figure 5.3.

<table>
<thead>
<tr>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolith [s]</td>
<td>37,122</td>
<td>19,098</td>
<td>10,057</td>
<td>5,731</td>
<td>4,708</td>
<td>3,385</td>
<td>3,555</td>
<td>3,024</td>
</tr>
<tr>
<td>Modular [s]</td>
<td>45,312</td>
<td>23,252</td>
<td>11,776</td>
<td>6,530</td>
<td>5,365</td>
<td>4,039</td>
<td>4,108</td>
<td>3,475</td>
</tr>
<tr>
<td>Overhead</td>
<td>22.1%</td>
<td>21.8%</td>
<td>17.1%</td>
<td>13.9%</td>
<td>14.0%</td>
<td>19.3%</td>
<td>15.6%</td>
<td>14.9%</td>
</tr>
</tbody>
</table>

Figure 5.3: Execution times for different number of workers, for both the monolithic and the modular version.

As expected, we see that the modular version is slower than its monolithic counterpart. Where the monolithic query uses in-memory queues to send data to the next stage, the participants of modular version have to serialize the messages and publish them in a topic. Both versions do not necessarily perform better if we use hyper-threading for additional worker threads, as 24 workers in both cases perform worse than 16.

Previous work has shown that sessionization is able perform real-time processing on this machine with 16 workers and more. Our monolithic version is indeed also able to finish the computation in under one hour with 16 workers and more. The modular version however only achieves execution times below the threshold of one hour with 32 workers. Further work has to be done to investigate where ex-
act the overhead comes from and optimize our system accordingly, such that we achieve real-time performance with less workers.

The relative overhead varies between 14-22%. It should be noted that relative overhead is not proportional to the number of worker threads. We will show in section 5.2.4 that other factors such as placement also impact the overhead.

5.2.2 Worker Utilization Factor

In this section we present the average measured utilization per worker, i.e. the proportion of time a worker thread performs useful work. This gives us an indicator of how well the different queries of the modular version are able to cope with the given load.

We define the utilization factor of a worker as the percentage of the time spent inside the Timely computation, i.e. utilization = \( \frac{\text{busy time}}{\text{total time}} \). To put it another way, a worker is not busy if it is waiting on external input. This is the case if it waits on input from a topic in the case of the subscribers, or if it reads data from disk in the case of the sessionization stage.

Figure 5.4 shows the utilization for the whole monolithic query compared to the sessionization stage of the modular query. The general trend for both versions is that utilization increases, i.e. the individual workers spend less time waiting on input from disk. This is to be expected, as with more workers, each worker has to read less input data. Note that the two versions do not perform the same amount of work: The monolithic query contains all stages of the dataflow graph, while the sessionization query contains only the first. With 8 and more workers, the modular version has a higher utilization. We assume that this is caused by the fact that the publisher starts to dominate the execution overhead.

Figure 5.5 shows the average utilization for the subscribers. In contrast to the first stage, their utilization decreases with more workers. Compared to Figure 5.4, we see that in all cases the average utilization in the sessionization workers is significantly higher than in the other queries of the modular version. This indicates that the sessionization stage itself is the bottleneck of the computation. We see that with a higher number of worker threads, individual workers are proportionally spending more and more time waiting for input.

This suggests that using the same amount of workers for all queries might not be the most efficient use of computational resources. Future experiments would have to explore how many worker threads are actually necessary for the different queries in order to keep up. This however also requires a more sophisticated assignment of topics to subscribing workers, which currently has to be manually encoded in the queries source code.
5.2 Results

Figure 5.4: Utilization of the monolithic version compared to the sessionization stage of the modular version.

Figure 5.5: Utilization for the individual subscribers in the modular version.
5.2.3 Memory Consumption and Processor Utilization

Figures 5.6 and 5.7 show the processor utilization and peak memory consumption for the different processes, as reported by the operating system. Given that the processes communicate over the loopback device, we do not measure the networking throughput.

Memory Consumption   Given the memory requirements sessionization task, we split Figure 5.6 into two graphs: The upper graph compares the sessionization query of the composed version with the memory consumption of the whole monolithic query, while the lower graph shows the memory footprint for the remaining queries.

We immediately see that the sessionization query of the composed version consumes the most memory. This is caused by the fact that the re-ordering and sessionization buffers incoming messages until certain epochs are completed. Also with regard to the sessionization query, we see that its memory footprint is higher than memory consumption for the monolithic version, even though it performs less work. We assume that this is caused by the queuing of outgoing messages at the publisher. As we add more workers, the consumers are better able to catch up and the queue shrinks.

Another contributing factor to the overhead is the fact we run additional operating system processes, each one with a large set of worker threads. This growth can be observed in the lower graph of Figure 5.6, as the memory overhead grows even though the amount of processed data stays the same.

Processor Utilization   We also report the overall processor utilization reported by the operating system. A value of 100% would correspond to the full utilization of all 32 logical cores. We see that neither versions manage to fully utilize the processors over the duration of the whole computation. This indicates that both versions are sometimes blocked on external input. This is to be expected, as the workers switch between reading input and processing the data within the same thread. Consistent with the observations from worker utilization, we also see that the sessionization stage is the most compute heavy.
Figure 5.6: Peak memory consumption of the different processes. The left bar in the upper graph shows the monolithic query, the right bar in the upper graph shows the memory consumption of the sessionization stage of the modular query. The lower graph shows the memory used by the other queries of the modular version.
Figure 5.7: CPU utilization for the different processes. The left bar shows the monolithic query, the right bar shows the processor utilization for the different query processes in the modular version.
5.2.4 Impact of Worker Placement

In the previous experiments, we prefer running as many workers as possible on a single processor socket (with the exception of hyper-threads). Because the modularized version has to copy the data from one process to the next one, and thus data might move from one socket to the other, we expect that the worker placement will have an impact on performance.

Figure 5.8 compares execution time of this strategy compared to a strategy where we balance the workers evenly between sockets. With 12 workers we are unable put all threads on a single socket, and we see that in this case it is slightly beneficial to balance the threads evenly.

The impact of using different placements is relatively small for the monolithic query. The modular version however is more sensitive to different placement strategies, resulting in changes to the overhead of the execution time.

![Execution Time Chart]

Figure 5.8: Execution times for different worker-to-processor mappings. The placement describes the number of worker threads placed on each of the two processor sockets.
5.3 DISCUSSION

In this chapter we presented an experimental evaluation of the overhead of query composition in our system. The results show that the modular version in our setting takes about 1-18% longer to execute and consumes 8-14% more memory. While this is not negligible, we are confident that a modular version of sessionization would be able to keep up with real-time processing, given enough resources.

We believe though that further experiments are indispensable to definitely answer this question. Unfortunately we were not able to conduct more experiments within the remaining time.

For one, future experiments should evaluate the performance of modular sessionization in a cluster of machines. We only ran our experiments on a single machine, meaning that data shared in topics was transmitted over the loopback interface. Having publishers on one machine and subscribers on another would certainly yield different performance characteristics, due to the network between them. The monolithic version would also likely show a different performance behavior, as workers would have to shuffle their data over the network instead of using shared memory queues.

In addition, we did not perform any microbenchmarks to evaluate the performance characteristics of other features of our system, such as the latency of spawning a new query or the time it takes to subscribe to a topic. While this would be interesting to evaluate, it currently would only be of little significance for long-lived queries such as sessionization, as we expect these features mostly to be used during the set-up phase of a query.
RELATION WORK

6.1 RELATED WORK

6.1.1 Dataflow Streaming Engines

In contrast to the standalone Timely Dataflow library, many state-of-the-art dataflow streaming engines have been designed and engineered from the start to run multiple concurrent dataflow computations in a cluster. One of the goals of this thesis was to provide such functionality for Timely Dataflow based programs. Thus, in this section, we will mainly focus on the execution and management aspect running multiple concurrent dataflow computations. Less focus is put on the differences in the programming and synchronization model these other systems provide. Our system currently also does not provide any fault tolerance guarantees. This is also a notable difference our system has compared to related work, which often put heavy emphasis on providing strong fault tolerance.

6.1.1.1 Spark Streaming

Spark Streaming [18] is a streaming engine based on the Spark cluster computation framework. Spark Streaming executes streaming dataflow computations by turning them into stateless, fault-tolerant batch computations. These batches are submitted as tasks to the underlying Spark engine. [17]

PROCESS MODEL  In Spark, computational tasks work is modeled as transformations on partitioned collections of records called resilient distribute datasets. The runtime schedules the execution of the transformations on a distributed set of worker nodes. By tracking the lineage of transformations, Spark is able to recover computations by reissuing lost transformations. Based on the model, Spark Streaming models stateful, conceptually continuous dataflow computations as small stateless microbatches. This model of execution is very different from Timely’s continuous operator model. In Timely, both the operator scheduling and the data of the computation are managed directly by the workers themselves, which are hosted by long living operating system processes. In Spark however, the data and code is managed by the system. Because of this, Spark is not only able to provide strong fault tolerance by reissuing failed computations, Spark is also able to load balance concurrently running streams and mitigate stranglers. While we currently do not offer these features, Timely does achieve
much lower latencies than Spark Streaming, which is mostly enabled by the fact that every Timely worker manages itself independently.

6.1.1.2 Storm & Heron

Heron [7] is the API-compatible successor of the Storm [15] streaming dataflow engine. Both Storm and Heron call their dataflow computations topologies, which are directed acyclic graphs of spouts (stream sources) and bolts (stream transformers). Parallelism is achieved by the user specifying the number of instances for each bolt and spout, as well as the partitioning strategy between them.

Process model The process model of Storm is very similar to our approach: A topology (roughly corresponding to a query in our system) is executed by multiple worker processes. These are operating system processes which are distributed over different machines, thus similar to our query processes. Every machine hosts a supervisor, which not only monitors the local worker processes, but also spawns new worker processes on behalf of the Nimbus, making Storm’s supervisors similar to our executors. The Nimbus is a central process to which new topologies are submitted, mirroring our coordinator.

Heron mostly differs from Storm in its internal architecture. Most notably, in Heron every spout or bolt now runs in its own operating system process. All processes belonging to the same topology are grouped together in an operating system container. The motivation for this change are reported to be better debug-ability and simpler resource management. The authors also state that their topologies seldom have more than three stages (i.e. bolts/spouts). We believe that such an architecture would not make much sense for Timely, as its operators are typically more numerous and more lightweight.

Another major change from Storm’s architecture is the fact Heron does not have a central coordinating process anymore. Being a single point of failure and a bottleneck, the Nimbus was considered to be a flawed design. Heron thus replaces the old responsibilities of the Nimbus by offloading them into a small set of independent processes. This might be something to consider for futures extensions to our coordinator.

6.1.1.3 Flink

Flink [4] is a dataflow streaming engine for directed acyclic dataflow graphs, however it does have support for iterative dataflows on the outermost level. Flink interleaves control events with data records. Control events are used for progress tracking and fault tolerance, which is done through snapshots. Progress tracking is implemented with global low watermarks, which denote the minimum timestamp
which can be emitted at the sources of the topology, enabling Flink to perform out-of-order processing.

**Process Model** The runtime architecture of Flink also uses a central process called the *job manager*, which accepts and manages computations submitted by the client. The job manager takes the user submitted code, translates it into an execution plan and potentially optimizes it.

The computations themselves are executed inside worker processes called *task managers*. Task managers provide a similar function as our executors, in that they provide access to computational resources and are distributed over potentially multiple machines. Like our executors, they can be dynamically added and announce themselves at the job manager. However, in contrast to our executors, task managers directly execute the operators and manage any data exchange between them. In our system, this task is done by the query library in conjunction with the Timely runtime.

For resource management purposes, each task manager provides a predefined number of task slots. These typically correspond with the number of CPU cores, though it is not enforced that a slot only uses one thread. The available memory is also distributed equally between the task slots of a task manager. The submitted dataflow computation is split up in multiple subtasks (one subtask typically corresponding to one operator), and each subtask is assigned to a task slot. Subtasks from the same submission can share a task slot if the operator supports it. This is used for example for pipelined operators which don’t require any exchange of data with operators running in other task slots. Because task managers can have multiple task slots, they can run operators from different computations within the same operating system process, similar to Spark Streaming. Unlike Spark Streaming however, these operators are continuous and do not migrate to other task slots during normal operation.

### 6.1.2 Sharing Dataflow Streams

In this section we discuss how composition of different dataflow streams can be achieved in other systems.

#### 6.1.2.1 Kafka

Kafka [6] is a distributed platform for accumulating and sharing streams. It provides a topic-based publish/subscribe service, where topics can have multiple partitions. Producers append their data to one or more partition, allowing subscribers to read from it. The data within a partition is stored persistently, allowing subscribers to read all previously published data sequentially and continue where they left off in case of failures. The data within a partition is ordered, how-
ever there is no defined ordering across multiple partitions of the same topic. Recent versions of Kafka also supports the assignment of event timestamps to messages.

**Integration with dataflow engines** Both Spark Streaming and Flink provide official connectors for Kafka, allowing dataflow computations to stream data from or to Kafka topics. Flink additionally supports the extraction of timestamps from topics. Similar to our system, Flink also provides a mechanism to exact watermarks from Kafka sources, allowing a partition to contain out-of-order data. Like our system, Flink is able to unify progress tracking information from multiple sources.

In some ways, Kafka is very similar to our publish/subscribe as we also expose potentially partitioned streams from which consumers, such as dataflow programs, can subscribe to. In contrast to Kafka, we have a strict one-to-one mapping between stream partitions and topics. Stream partitions in our system are only grouped together by naming conventions on the topics.

Another difference between Kafka and our system is that Kafka’s subscribers are pulling data from their source, whereas in our system data is pushed from the source to the subscribers. This implies that in our system the publisher does not know about the progress of the subscriber, which can lead to backpressure issues if the subscribers are slow.

**Kafka Streams** In additions to the above mentioned adapters, Kafka also provides its own streaming engine called Kafka Streams. It provides a simple acyclic dataflow model which uses Kafka topics as sources and sinks, thus acting as transformers. Parallelism is achieved by instantiating the dataflow graph on multiple threads. Partitioning of the data happens before it is fed to the individual instances of the dataflow graph.

Like standalone Timely Dataflow, Kafka Streams is implemented as a library. It is left to the user to deploy and launch instances of the streaming computation.

6.1.2.2 *MillWheel*

MillWheel [2] is a stream processing framework used to implement the dataflow model proposed by Google [3]. MillWheel conceptually treats the whole system as a single dynamic dataflow graph: The nodes of the dataflow graph are user submitted computations (i.e. operators) that are invoked by the system on receipt of incoming data. Edges are created by the user specifying which streams a computation consumes and produces. Streams have uniquely assigned names.

This makes MillWheel essentially a publish/subscribe system, as every individual operator acts as a subscriber and as a publisher.
This is slightly different from our system and other combinations of dataflow systems that use an external publish/subscribe mechanism: Streams between operators in our system are anonymous until explicitly published. In MillWheel however all streams used to connect operators are automatically made available for subscription by third parties.

Another difference to our approach is the fact that MillWheel lets subscriber specify how a stream should be partitioned before it is delivered to the consumer. Aggregation and partitioning is done according to keys, i.e. records with the same key are always delivered to the same computation instance. Different computations can however use different keys on the same streams, as every computation needs to provide a key extraction function for each of its input streams.

Computations run on one or more processes distributed over a dynamic set of machines. The assignment of computations to machines and keys to computations is managed by the system itself. Because the system manages persistent state on behalf of the computations, computations for a certain key can be moved for load-balancing or restarted in the case of failures.
7.1 FUTURE WORK

7.1.1 Extracting the Dataflow Graph from Queries

Our system currently treats queries for the most part as black boxes. While we do dynamically track some information about the query, such as its publications and subscriptions, the system does not know anything about a query’s internal dataflow graph. The reason for this is discussed in Section 2.2.3 Runtime Graph Representation: Timely itself only assembles a type-erased representation of the dataflow graph during execution. The integrated logging framework also only exposes the structure of the dataflow graph, but not any properties about the nodes and edges themselves besides the name of the operators. While further instrumentation of Timely would certainly be able to provide more insight into the dataflow graph, solely run-time oriented approaches are limited in what they can do: Rust does not provide any run-time reflection of types besides unique type identifiers, and most operators accept user-defined functions to implement parts of their logic. We therefore believe that some form of static analysis is unavoidable for extracting precise information about the dataflow graph. Possible uses for more precise descriptions of the dataflow graph of a query are described below.

7.1.2 Retroactive Tapping of Dataflow Edges

With the current publish/subscribe mechanism, query authors are required to anticipate which operator outputs are of possible interest for subscribers. If a certain output is not explicitly published through a publish operator, there is no way for other queries to access it. We believe that some mechanism for retroactively exposing dataflow edges in a running query would be useful. Possible use-cases include better diagnostics, more flexible query composition, and potentially also query optimization.

We considered implementing such a feature as part of this thesis, and the design choice of topics that can be dynamically added or removed from the catalog was explicitly made with such a use-case in mind. However, due to time constraints, we were not able to pursue an implementation. In the remainder of this section, we will however present some of our preliminary ideas.
exposing stream handles  We believe that it would be relatively easy to instrument Timely such that it exposes its data streams for later use without much run-time overhead: Timely’s stream handle internally maintains a registrar which is used by succeeding operators to register themselves as consumers. When data is sent to an operator’s output, the data is pushed one by one into the queues of the consumers that have registered themselves on that output. By collecting and exposing the registrars of all edges, it would easily be possible to add new consumers at run-time. An unresolved issue with this approach however is that it would only allow the inspection of the “data plane” of a Timely computation. Because progress tracking information is delivered separately, a way has to be found to expose this additional information as well.

processing intercepted streams  Previous work on monitoring distributed systems such as P2 [14] and Pivot Tracing [9] has demonstrated powerful interfaces for processing and querying intercepted streams of running systems. We believe that a functionality to tap into dataflow edges of running queries would enable similar possibilities in our system: By publishing tapped dataflow edges as topics, other queries could be used to monitor, diagnose, or extend a running dataflow computation.

Furthermore, the overhead of serializing intercepted tuples could be avoided by extending the system to dynamically load code into running queries. Systems such as Pivot Tracing are using Java’s ability to dynamically load bytecode into the running programs. For our Rust-based system we envision a mechanism where queries are attached as dynamic shared objects onto other running queries.

work deduplication  A detailed representation of a query’s dataflow graph, together with the ability to tap into dataflow edges, could also be used for query optimization. An example of this is dataflow path deduplication: If a new incoming query performs the same sequence of operations on the same stream of data as an already existing query, we would like to deduplicate this work. By exploiting the fact that topics act as semantic descriptions of streams, the system could automatically tap into dataflow edges on the outgoing path of a publication in order to optimize the subscriptions of submitted queries. Such an optimization however requires that the system is able to compare the semantics of operators, i.e. it needs to be able to determine if two operators are equivalent. It would also require a way for the system to remove or insert operators before executing the query. An example of such an optimization is shown in Figure 7.1.
Figure 7.1: Work duplication through derived topics. By tapping into data-flow edges of the publishing query, the system can optimize the newly arrived subscribing query.

7.1.3 Resource Management

For the most part, resource management is still an unresolved problem in our current system. One aspect of this is the placement of queries on the available executors. It is currently the responsibility of the user to select which executors will host a new query submission. The reason for this is that the system does not know what conditions have to be fulfilled in order to execute the query: The query might access external files or other resources which are currently not expressed in the system model through the catalog.

However, in many cases automatic placement of queries might be desired. Future work would have to explore what kind of information a query placement scheduler requires to perform well, and extending executors or other components to collect this kind of information.

Another aspect of resource management is resource control. Once a placement for a query has been selected, the responsible executors might want to ensure that the available resources are distributed fairly among the running queries. Currently, CPU and memory usage of queries is solely managed by the operating system. The use of operating system features such as Linux control groups [12] could be added to more precisely indicate to the operating system what kind of resources certain queries should have access to.

Because queries can run arbitrary code and might not be trusted, future work might also want to investigate sandboxing or other protection techniques to further isolate queries from each other and the underlying operating system.
7.1.4 Backpressure, Buffering and Persistence

Our current system does not buffer data at the publisher if there are no subscribers. It does however buffer outgoing data for connected subscribers until they consume it or disconnect. This can become an issue if a single subscriber is slower than the publisher, as its queue will grow and thus memory consumption increase. Future work has to explore how to throttle the publishing query in such situations. One possible approach would be the addition of probes which indicate the subscribers progress. These would then be used to throttle the input of the publishing query. Previous work [8] has successfully demonstrated the use of such a flow control mechanism within the scope of a single query, future work would have to explore how to integrate this in our system.

Tightly coupled queries can already implement such an approach today, as our system allows two queries to subscribe to each other: Besides the topic published by the producer and read by the consumer, there would be an additional second topic published by the consumer and read by the producer used as a back channel for progress probing.

Another approach to deal with slow subscribers could be the introduction of intermediate buffers, which store the published data in persistent storage. This would reduce memory pressure in the presence of slow subscribers and it would also allow late subscribers to replay the whole data stream, if such a feature is desired.

Intermediate buffers could also be used to reduce network traffic if many co-located subscribers consume data from same remote publisher: By having subscribers read their data from local buffers instead of the remote publisher, data would only have to be sent once over the network.

7.1.5 Partitioning and Filtering by Subscribers

In our current system, the partitioning scheme of published streams is determined by the publishers, which either publishes one topic per worker or a single topic for all workers. If a subscribing query uses a different amount of workers than there are available topics, the query author of the subscribing query has to manually distribute these topics among the query’s workers.

MillWheel by contrast allows subscribers to provide their own partitioning functions which are used by the system to automatically decide how data is routed from the publishers to the subscriber instances. It is not completely clear how such a functionality could be implemented in our system, as we currently do not have intermediate message brokers which could do the routing.
One possible implementation approach could however work by allowing subscribers to tell the publishers about the kind of data they are interested in. Custom routing functions would then be implemented by the subscribers connecting to all the publisher partitions, but only receiving a subset of the published data based on the provided filters. This approach would be similar to content-based filtering and routing found in other publish/subscribe systems [5].

7.2 CONCLUSION

In this thesis, we have presented a system for deploying Timely Dataflow applications. Our approach is based on the assumption that individually submitted queries are not developed and executed in isolation, but rather that they are small parts of a larger overall application. For this reason, we focused not only on how to share common input sources, but also on how queries can share their results with their peers.

We presented an implementation which can deal with the concurrent submission of queries, allows the dynamic addition of new machines to the cluster, and permits queries to inspect the current system state. We extended Timely Dataflow with a publish/subscribe mechanism that integrates Timely’s progress tracking mechanism and enables the composition of queries.

We demonstrated how this manual query composition performs on a realistic workload. While our evaluation shows that the overhead is not negligible, we are confident that further investigations and subsequent optimizations can successfully reduce this overhead.

Our system was designed with future extensions in mind. We shared some preliminary ideas how data streams of running queries could be exposed retroactively, to allow more flexible query composition. We also expect that the overhead of serialization in our publish/subscribe system could be avoided by having the executor load the subscribing queries in the address space of the subscriber.

We believe that our work, while by itself not particularly novel in its individual features, is a solid building block which will enable the development and implementation of future Timely Dataflow based applications.


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

An Online Stream Processor for Timely Dataflow

Author(s) by (in block letters):

For papers written by groups the names of all authors are required.

Name(s): Wicki
First name(s): Sebastian

With my signature I confirm that
− I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
− I have documented all methods, data and processes truthfully.
− I have not manipulated any data.
− I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, 13.11.2016

Signature(s)