UNIVERSITY OF OSLO
Department of Informatics

Creating a GStreamer plugin for low latency distribution of multimedia content

Master thesis

Ján Vorčák

May 6, 2014

Acknowledgement

I would like to thank my supervisors, Thomas Plagemann and Hans Vatne Hansen for their invaluable feedback and guidance. Thank you for the supervision of the thesis and for the opportunity to work on this project.

I am most thankful to my family and friends who supported my studies and all the decisions I have made. I would like to especially thank Marta Lajtová for her patience and support during my studies.

Ján Vorčák
University of Oslo
May 2014

Abstract

Electronic devices such as personal computers, tablets and smart phones have recently become more available for the majority of users. These devices differ in multiple aspects including their screen size, processing power and other properties like mobility and usability. Since each of these devices has different advantages, these computing devices can collaborate and share data to provide a better user experience. By separating an application into several components, we are able to run these components on different devices where they can perform best.

We have designed, implemented and evaluated a development tool for creating distributed applications which run in a community of heterogeneous devices. When it comes to popularity and user's demands, multimedia applications are also becoming an essential part of systems used on personal devices. We have therefore focused our work on processing multimedia content. Even though our tool is tailored to the needs of multimedia applications, it can handle any type of data stream.

Several collaboration platforms have been proposed that allow data sharing in a distributed environment. We have analysed these platforms with respect to their latency and throughput. Based on this analysis, we have chosen the one which is optimized to work with multimedia - *TRAMP Data Exchanger*.

In order to make development of such an application faster, we have analysed several multimedia frameworks that can provide a higher abstraction for this collaboration platform. The role of a multimedia framework is to provide an intuitive API for processing multimedia content on a computer and through a network. We have chosen one of them - GStreamer to serve as a wrapper for underlying the propagation system. We have designed, implemented and evaluated a GStreamer plugin which wraps the functionality of the *TRAMP* data sharing platform. When using this plugin, it is easy for application programmers to develop multimedia applications distributed over several devices while taking advantage of the powerful GStreamer features and plugins. Since the concept of *TRAMP Data Exchanger* is a work in progress, our GStreamer plugin also helps to test and evaluate this collaboration platform.

We do evaluation by using measurements in a distributed environment. We run these measurements on various hardware on top of different operating systems to demonstrate the ability to run on heterogeneous devices.

Our evaluation shows that when using our tool, the development process is

easier and faster for application programmers while keeping the requirements for multimedia applications within the acceptable levels. The latency overhead introduced by our system is under 100 milliseconds, which is acceptable even for real-time multimedia applications. We have managed to send various multimedia streams at different bit rates up to 5Mbit/s. Using these experiments, we have demonstrated that our plugin is usable in the context of multimedia applications and consequently allows them to run in a distributed environment.

Contents

Acknowledgement **1**

Abstract **3**

1 Introduction **15**

1.1 Background 15
1.2 Motivation 16
1.3 Requirements 17
1.4 Outline 18

2 Background and Related Work **19**

2.1 Munin 19
2.2 Linda 21
2.3 TRAMP Data Exchanger 22
2.3.1 Application Component API 23
2.3.2 Internal messages 24
2.3.3 Multimedia content 25
2.3.4 Architecture of TRAMP Data Exchanger 25
2.4 GStreamer 29
2.4.1 Foundations 29
2.4.2 Elements 29
2.4.3 Plugins 31
2.4.4 Pads 31
2.4.5 Properties 31
2.4.6 Communication 32
2.4.7 Capabilities negotiation types 32
2.4.8 Capabilities negotiation process 33
2.4.9 Renegotiation 34
2.4.10 Structure of GStreamer elements 35
2.5 VLC media player 37

3 Design 41
3.1 Goals . . . 41
3.2 Data propagation framework . . . 41
3.3 Multimedia framework . . . 43
3.3.1 Overview . . . 43
3.3.2 GStreamer . . . 43
3.3.3 VLC . . . 44
3.3.4 Choosing the multimedia framework . . . 44
3.4 Detailed design . . . 44
3.4.1 TRAMP plugin . . . 45
3.4.2 Architecture overview . . . 46
3.4.3 TRAMP GStreamer elements . . . 46
3.4.4 Labels . . . 47
3.4.5 Caps negotiation . . . 48
3.4.6 Communication between pipelines . . . 49
3.4.7 Proxy implementation . . . 53
3.4.8 Data replication . . . 54
3.4.9 Using multiple TRAMP Source elements . . . 55
3.5 Analysing TRAMP Data Exchanger . . . 56
3.5.1 Updating peers with inconsistent data . . . 56
3.5.2 Rebalancing distribution trees . . . 58
3.5.3 Subscribing to the data segment that is not yet published . 59
3.5.4 Copying memory to the shared data segment . . . 60

4 Implementation 61
4.1 Overview . . . 61
4.2 Implementation details . . . 62

5 Evaluation 69
5.1 Evaluation goals . . . 69
5.2 Evaluation approach . . . 70
5.3 Evaluation factors . . . 71
5.3.1 Node specifications . . . 71
5.3.2 Network speed . . . 71
5.3.3 Maximum buffer size of TRAMP Data Exchanger . . . 72
5.4 Metrics . . . 72
5.4.1 Latency . . . 72
5.4.2 Processed buffers per time interval . . . 75
5.4.3 CPU load . . . 76
5.4.4 Memory load . . . 76
5.4.5 Comparison with the UDP plugin . . . 78
5.5 Results . . . 79
5.5.1 Latency . . . 79
5.5.2 Processed buffers per time interval . . . 80

5.5.3 CPU/Memory load 82
5.5.4 Comparison with the UDP plugin 84
5.5.5 User friendliness 85
5.6 Discussion 87

6 Conclusions **89**
6.1 Contribution 89
6.2 Future work 90

Bibliography **91**

Glossary **95**

APPENDICES **95**

A Deployment **97**
A.1 Compiling RPM packages 97
A.2 Installation 98
A.3 Usage 98

B Source code **99**

List of Figures

1.1 Simple Raspberry PI camera system 16

2.1 Building distribution trees in TRAMP Data Exchanger 24
2.2 TRAMP Data Exchanger architecture overview 26
2.3 Detailed design of TRAMP Data Exchanger 27
2.4 Connection thread functionality 28
2.5 Example of GStreamer pipeline 30
2.6 Example of GStreamer pipeline with different value of a *videotestsrc pattern* property 31
2.7 Example of a GStreamer pipeline with different communication flows 32
2.8 Negotiation between two elements 34
2.9 Renegotiation between two elements 35

3.1 Architecture of our system 45
3.2 Architecture overview 46
3.3 Example of two GStreamer pipelines interacting using TRAMP daemon 47
3.4 Using GStreamer labels with TRAMP Data Exchanger 48
3.5 Using additional shared data segment to share the capabilities information 50
3.6 Internal buffer used by our plugin 51
3.7 Proxying events & queries 52
3.8 Proxy implementation 54
3.9 TRAMP: Example of data propagation with replication 55
3.10 TRAMP: Example of data propagation without replication 55

5.1 Measuring the Round trip time 73
5.2 Measuring latency on two machines with synchronized clocks . . 74
5.3 Setup for measuring CPU/Memory load 77
5.4 Setup for comparing the plugin to the UDP Plugin 78
5.5 Latency 79
5.6 Different miss values depend on the value of MTU 80
5.7 Missed data segments with different values of MTU property . . . 81

5.8 Number of missed data segments with size of the shared buffer . . 82
5.9 Number of missed data segments - comparing nodes with different performance capabilities 83
5.10 CPU load of the consuming/replicating process 84
5.11 Memory load of the consuming/replicating process 84
5.12 Distribution tree built with TRAMP 84
5.13 Network history of the consumer - replicator 85

List of Tables

2.1 Linda API 21
2.2 TRAMP Application Component API 23
2.3 Internal message structure 29
2.4 List of GStreamer element internal functions 36
2.5 VLC module types 38

3.1 Comparison of data sharing platforms 43
3.2 Important events to proxy 51
3.3 Important queries to proxy 53
3.4 Structure t_message 53
3.5 Sending capabilities query downstream 54
3.6 TRAMP API extension 59

5.1 Node specifications 71

List of Listings

2.1 Release consistency 20
2.2 Behaviour of out function in Linda 21
2.3 Publishing data segment using TRAMP Data Exchanger 23
2.4 Retrieving data segment using TRAMP Data Exchanger 23
2.5 Choosing a specific video output in VLC 37
2.6 Code structure of VLC module 38
3.1 Setting a property on the sink element 48
4.1 Defining constants and types 62
4.2 Initializing meta data and function pointers 63
4.3 Initialization of a new sink element instance 64
4.4 Implementation of gst_trampsink_set_caps function 64
4.5 Implementation of gst_trampsink_render function 65
4.6 Implementation of gst_trampsrc_render function 66
4.7 Implementation of gst_trampsink_render function 66
4.8 Implementation of gst_trampsink_render function 67
4.9 Implementation of gst_trampsink_render function 67
5.1 GStreamer pipelines used in Setup #1 86
5.2 GStreamer pipelines used in Setup #2 86
A.1 Running GStreamer TRAMP plugin 98
A.2 Installing TRAMP GStreamer plugin on Fedora 98
B.1 Downloading our version of TRAMP Data Exchanger 99
B.2 Fetching the source code of GStreamer TRAMP plugin 99

Chapter 1

Introduction

1.1 Background

In today's world, we are surrounded by an enormous number of heterogeneous devices. Devices such as personal computers, laptops, tablets and smart phones have increased in popularity and availability. These devices differ in their capabilities, processing power, screen size and other properties like mobility and battery life. That means we are no longer forced to run one application per device, but we can think of taking advantage of several devices we own and use those devices to serve our applications.

We can for instance control a presentation or video conference with an easy-to-use tablet, while still having a presentation displayed on a high resolution LCD screen. For this to be feasible an application needs to be separated into several parts that are able to run on different devices, these are called the components. One or more collaborating components can form an application. The components need to communicate and exchange data amongst themselves. The main benefit of splitting an application into several components is that each component can run on the device where it can perform the best.

There is a need for a collaboration platform that can handle a wide range of applications - starting from simple applications that need to have their components running on several devices ending with real-time distributed multimedia applications that have strict deadlines. Programmers should be able to work with memory distributed over several devices where the application's components run.

Distributed Shared Memory (DSM) is an abstraction where physically separated parts of the memory can be accessed as if they are part of one logical address space. While there are several hardware systems which work on the architectural level, we focus on software DSM systems which require a runtime system to work. Focusing on software DSM systems allows us to write multi-platform applications which are easy to install on several stand-alone devices.

1.2 Motivation

When writing an application distributed over multiple nodes, it would be beneficial for programmers to have an abstraction of linear address spaces so they can focus on the application's logic, instead of the underlying data propagation. There should be easy-to-use API available for programmers, so that they do not need to know where the data is located or in what way the data is distributed among devices. While some DSM systems provide programmers with friendly API that fulfill these requirements, there is still a possibility to provide programmers with even higher abstraction. Since multimedia applications have recently increased their popularity, we have decided to make it easier for application programmers to use these DSM systems in their multimedia applications. We have chosen one DSM system which is the most suitable for writing multimedia applications and integrated it with a multimedia framework by developing a plugin for it. Once the DSM system is integrated with the multimedia framework, application programmers are provided with powerful tools of multimedia framework while working with the distributed shared memory.

Our multimedia plugin allows application programmers to easily write multimedia applications distributed over several machines. We claim that such applications enhances the overall user experience.

Accessibility of today's cheap hardware [6] with the combination of this easy-to-use multimedia plugin makes it easier for developers to create distributed multimedia applications. An example of such an application is a simple camera system taking advantage of today's minicomputers illustrated in Figure 1.1. A set of minicomputers can interact with each other by recording and sharing various multimedia streams e.g. video captured by a camera. These streams can be played in real-time or stored on other devices within this system. With our plugin, such a system can be easily set up using command line tools only, with no need to write a single line of code.

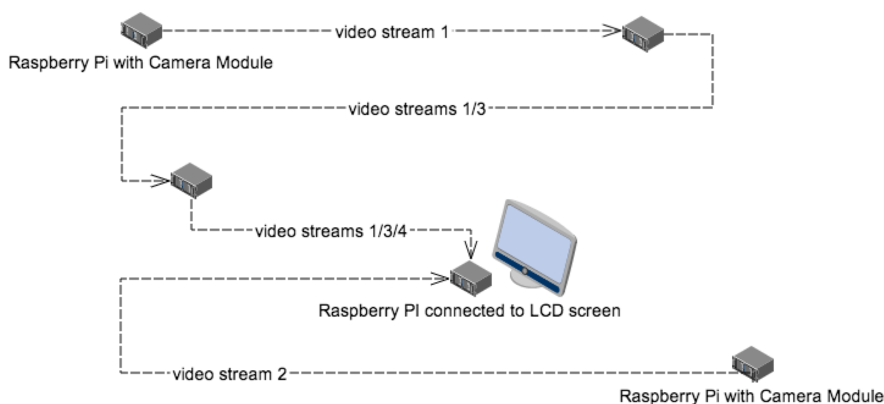


Figure 1.1: Simple Raspberry PI camera system

Other possible use cases can be found everywhere where streaming/multimedia content needs to be transmitted within a group of devices e.g. video chat applications, gaming devices, etc.

1.3 Requirements

Our main goal is to determine if our system can be used for real-time multimedia applications running on multiple devices. We present the following specific requirements for our system:

1. **Performance** Our plugin should able to handle any type of data stream, but it should be tailored to the needs of multimedia applications. It is necessary for our system to fulfill these requirements in terms of latency and throughput.

   For real-time applications such as gaming or video conferencing, we want to keep latency as low as possible. In order to support these kind of applications, we want to keep the latency overhead less than 100 ms [13].

2. **Multi-platform** Since we want to support as many devices as possible, our system should not be bound to one specific operating system or architecture.

   In the future, components will be able to move from one device to another. Therefore, our collaboration platform should not be bound to one operating system. It should ideally support personal computers as well as mobile devices.

3. **Ease of use** It should be possible to create multimedia applications running on several devices without the need to take care of underlying data propagation. The knowledge of a chosen multimedia framework should be sufficient for a programmer to set up a distributed multimedia application.

4. **Minimal overhead** in terms of scalability. Our system should support the same number of devices as an underlying DSM system. It should not decrease the scalability of this underlying system, so the application programmer does not need to trade simplicity of code for scalability.

Our plugin serves as a wrapper for an underlying DSM system to make the programmer's work easier. The component which takes care of data propagation is the DSM system. It is therefore crucial that it is fast, efficient and stable.

In consequence to this requirement, another goal is to analyse the design and code of the underlying DSM system to ensure it fulfils these requirements. The multimedia plugin also serves as a tool to easily analyse this DSM system with real multimedia data.

1.4 Outline

This thesis is organized as follows: Chapter 2 gives an introduction to the DSM systems and multimedia frameworks we considered to use as part of our system. Chapter 3 describes our design goals. Based on these goals we chose the underlying DSM and multimedia frameworks. At the end of the chapter, we provide an in-depth analysis of chosen DSM framework with respect to our requirements. In Chapter 4, we write about the implementation of our system. The implementation is evaluated in Chapter 5 with reference to the requirements. We conclude our work and discuss issues to be solved in the future in Chapter 6.

Appendix A contains information about the deployment to RPM-compatible platforms. Appendix B provides the link to the git repository along with important instructions on dependencies and system configuration.

Chapter 2

Background and Related Work

In the first part of this chapter we analyse DSM systems providing data in the distributed environment. These include Munin discussed in Section 2.1, Linda in Section 2.2 and TRAMP Data Exchanger in Section 2.3. The second part of this chapter contains the analysis of multimedia frameworks - GStreamer in Section 2.4 and VLC in Section 2.5.

There have been several DSM systems proposed. They differ in speed, the way data is propagated to other devices and in the way they guarantee consistency. There are several optimization techniques possible. One approach is to replicate data. Another is to not share the whole address space, but to select data we want to share - usually by letting the application programmer to annotate shared variables [29].

For an application that takes advantage of several devices and is working with multimedia data, we require shared memory systems to be fast, non-blocking and with minimal latency.

Since in our case, applications running their separate components on several devices are expected to run on a small number of personal devices, a DSM system does not have to be highly scalable.

There are several DSM systems that are trying to solve this issue.

Some systems like Munin [3] or Midway [4] use specific variables that are shared among different systems. Others like Linda [11] are object based, so that the system offers a higher degree of abstraction and runtime control, over what is shared.

2.1 Munin

Munin is a DSM system based on sharing software variables, which are explicitly annotated by the **shared** keyword.

At start-up, Munin root thread registers itself with the kernel as the address space page fault handler. Therefore programmers can access these variables normally using read [23]/write [24] calls. In case any data is not locally accessible, retrieving

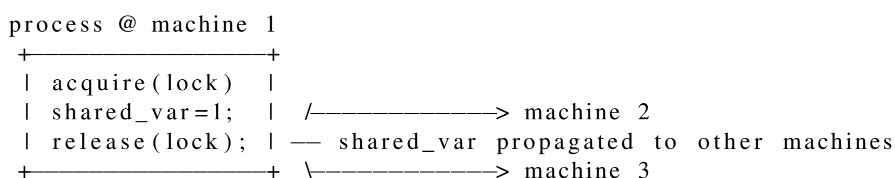
data is handled by a page fault handler. In order to guarantee consistency and minimal bandwidth, Munin uses release consistency [12] and supports four types of shared data variables.

- read only - immutable variables
- migratory - one thread can access the shared object at a given period of time
- write-shared – the shared object can be written by multiple threads at the same time
- producer-consumer – the shared object has one producer generating data and several consumers reading the variable

The type of the shared variable is determined from the annotation that is provided by the application programmer for each data type. Using the *producer-consumer* variable type is the most suitable approach for writing distributed multimedia applications running on several devices. Multimedia content is usually produced by one node only while being consumed by several nodes at the same time. This protocol guarantees to update the object instead of invalidating it. That causes elimination of read misses by consumer processes. Until producer process releases the lock, all changes to these objects are buffered.

Release consistency model As stated above, Munin uses release consistency model to guarantee consistency in a distributed environment. This consistency model provides us with two accesses - *acquire* and *release* as seen in Listing 2.1. *Acquire* is used to notify the memory system that a calling process is about to enter a critical section. *Release* says that a calling process has left a critical section. After *acquire* it is successfully completed, changes to the shared variables can be done locally. At this time, changes are not propagated to other machines. Once the machine has finished processing the critical section, it calls *release*, which causes all changes to the shared variables being propagated to other machines.

Listing 2.1: Release consistency



```
process @ machine 1
+------------------+
| acquire(lock)    |
| shared_var=1;    |  /--------------> machine 2
| release(lock);   | -- shared_var propagated to other machines
+------------------+  \--------------> machine 3
```

Release consistency has a lot of advantages in a distributed environment. It postpones propagation of changes to the shared variables until *release* it has finished. This decreases the number of messages sent among machines. However once release has finished, all messages are propagated to all other machines in the system. Lazy

release consistency [15] is the improvement of release consistency which lazily pulls modifications only when they are needed by other machines instead of pushing them to all machines by the machine completion of the *release* call.

Release consistency model used by Munin is useful for parallel programs where it is important to decrease the number of messages sent among nodes, thus decreasing the bandwidth. However, this is achieved at the cost of higher latency which is not acceptable for multimedia applications.

2.2 Linda

Linda [11] is a DSM object based system which allows us to share data tuples similar to *tuple* [9] in *Python* [8] or *struct* in *C*. These tuples are put into one tuple space which is shared by all nodes in the system. Programmers are provided with six operations shown in Table 2.1. Searching for available data tuples is done based on the type and the content of the elements in the tuple.

in	removes a tuple from the tuple space
out	places a tuple into the tuple space
rd	get a tuple from the tuple space without removing it
inp	non-blocking version of in
rdp	non-blocking version of rp
eval	evaluate function passed as an argument and places its return value to the tuple space

Table 2.1: Linda API

If there is no available tuple in the tuple space, the process blocks and waits for another process to insert a corresponding tuple to the tuple space. This blocking behaviour does not suit multimedia real-time applications very well, because of the increased latency. We could avoid blocking by using non-blocking functions *inp* and *rdp*. As seen in Listing 2.2 *out* function is asynchronous. After successful execution of *out* function, we are not guaranteed that the value is in the tuple space.

Listing 2.2: Behaviour of out function in Linda

```
1 out("example_tuple", 55)
2 ret = rdp("example_tuple", ?x) // x can be 55 or NULL
3 in("example_tuple", ?y) // y will eventually be 55
```

There are several Linda based implementations in various programming languages. Some of them like JavaSpaces [10] are not suitable for our application, because of their centralized design. One node is usually responsible for all data.

Since our application operates in personal device federations, this design is not suitable because of inherent churn in this environment.

Another approach of storing tuples in Linda is the broadcasting of tuples to all nodes. The S/Net Linda [7] system uses this implementation. Each node stores tuples locally and every operation is broadcasted to every other node in the system. As already stated above in this thesis, our aim is to handle multimedia content with high throughput and low latency. Design based on broadcasting every data chunk does not scale very well and would drastically increase bandwidth in the environment.

Another solution is presented in [16]. In this implementation, tuple space is divided into disjointed subspaces which are distributed over all of the nodes in the system. Nodes are logically organized in a grid. Once *out* is performed, tuple is broadcasted to all of the nodes in the current row of the current machine. When *in* call is performed, the template is broadcasted to all of the nodes in the current column. The node which is included in the row as well as in the column, handles the request. Since we want to achieve low latency for our system, broadcasting the template every time we want to get the data segment is not efficient.

2.3 TRAMP Data Exchanger

Another system for sharing memory segments called *TRAMP Data Exchanger* has been proposed as part of the TRAMP project [13], which uses distribution trees to share data among different devices. Data segments in this system are associated with labels, so that each of the shared data structures has its own identifier. The system automatically builds distribution trees based on latency. Distribution trees are built per data segment.

Each of these shared data segments can have one producer generating data and one or more consumers. The system is meant to work in small federations of personal devices and trades off performance for scalability. Devices in these small federations are organized in a full mesh topology. In a full mesh topology, there are direct connections between all connected devices, thus we gain high performance. That is important if we want to support multimedia applications with a high data transfer rate and low latency requirements. On receiving a data segment, consumers also act as replicators of the data and inform all other nodes about this fact.

This design has several advantages, mostly for sharing multimedia data. Since consumers also replicate data, each node can subscribe to get the needed data segment from the node with the lowest latency for this data segment.

The system uses broadcasting to inform other nodes what type of data segments it can provide. The system is meant to minimize latency, it does not scale very well because of the full mesh topology. When used in the federation of personal devices, scalability of this solution is sufficient. Since distribution trees are built per data segment and are built when the data segment is requested, they are not very adaptive to latency changes in the network.

2.3.1 Application Component API

From the application programmer's point of view, *TRAMP Data Exchanger* provides us with API containing four functions for handling shared data segments listed in Table 2.2.

Initialize	Association of data segment with a label
Publish	Publishing data segment for all components
Get	Getting a reference to the data segment
Subscribe	Subscribing to data segments updates

Table 2.2: TRAMP Application Component API

Data segments in *TRAMP Data Exchanger* are associated with labels. Let's suppose we have several machines connected together using *TRAMP Data Exchanger* as illustrated in Figure 2.1. We can publish a data segment on one machine with label *AUDIO_BUFFER* using the code in Listing 2.3.

Listing 2.3: Publishing data segment using TRAMP Data Exchanger

```
1 char* audio_label = "audio_buffer";
2 char *buffer = \
3         (char *) tramp_initialize(audio_label, BUFFER_SIZE);
4 tramp_publish(audio_label, BUFFER_SIZE);
```

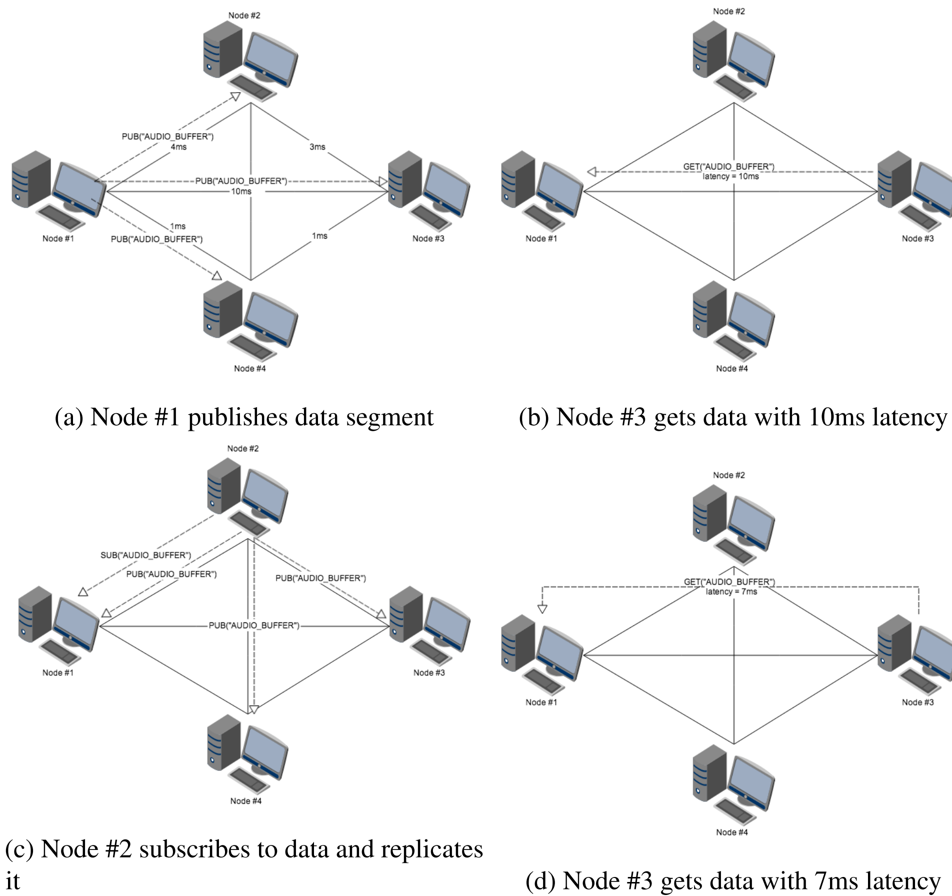
Listing 2.4: Retrieving data segment using TRAMP Data Exchanger

```
1 char* audio_label = "audio_buffer";
2 char *buffer = \
3         (char *) tramp_initialize(audio_label, BUFFER_SIZE);
4 tramp_get(audio_label, BUFFER_SIZE);
```

On all of the other machines we can access this buffer using *tramp_get* or *tramp_subscribe* as shown in Listing 2.4.

The framework takes care of building underlying distribution trees based on latency. Listing 2.3 shows the code which causes the behaviour illustrated in Figure 2.1a. *Node #1* publishes the data segment with label *AUDIO_BUFFER*. In this case, if *Node #3* wants to get this data segment using *tramp_get* or *tramp_subscribe*, it gets the originator of the data segment (*Node #1 in this case)* and all the replicators (no replicators in this case) and choose the node with lowest latency for this data segment. It therefore gets the data segment from *Node #1* as illustrated in Figure 2.1b, because it's the only node that owns this data segment.

Let us consider another situation illustrated in Figure 2.1c. *Node #2* decides to subscribe for the data segment *AUDIO_BUFFER*. Since it subscribes to this data



(a) Node #1 publishes data segment

(b) Node #3 gets data with 10ms latency

(c) Node #2 subscribes to data and replicates it

(d) Node #3 gets data with 7ms latency

Figure 2.1: Building distribution trees in TRAMP Data Exchanger

segment, it is updated on every data segment memory change. That allows it to become a replicator for this data segment. Therefore it broadcasts this information to all nodes using *tramp_publish* call.

After *Node #2* became a replicator, *Node #3* no longer asks *Node #1* for data. It considers the originator of the data segment (*Node #1*) and all the replicators (*Node #2*) and chooses the one with lowest latency. In consequence it gets the data from *Node #2* with a latency of 7ms.

2.3.2 Internal messages

API calls mentioned above are visible to the application programmer. Inside the middleware, there are six control messages taking care of the underlying data propagation and defining the system behaviour. API calls visible to the application programmer wrap these internal messages so that application programmer does not need to have knowledge of the framework's internal calls.

In the following paragraphs, we list and describe internal calls of the framework.

PUB: Sender informs receiver that it can deliver the data segment identified by a label passed as an argument. Receiver stores this information about availability of the data segment if it has not done so already.

LOOKUP: Sender checks whether receiver owns the data segment identified by a given label. Receiver first checks if he is the producer of the requested data segment. If the data segment is produced by the receiver itself, it sends corresponding *YEP* message with delay set to 0. If the receiver is the replicator, it replies with the *YEP* message with the information about the latency from the original producer.

YEP: *YEP* is an answer to the *LOOKUP:* message, by sending *YEP*, node informs the receiver that he owns the data segment with a given label. It also informs the receiver about inherited delay - time that it takes him to retrieve the data from the producer. This delay is equal to 0 in case the node is the originator of the data segment.

GET: The sender asks the receiver for the data segment identified by a given label. The receiver answers with *DAT* a message containing the actual data. If the receiver has no data associated with a given label, it does nothing.

SUB: The sender asks receiver to continuously send him updates of the data segment. The receiver creates a new thread for this purpose. The created thread continuously checks for memory changes in the data segment. If the data segment is changed, it sends *DAT* a message to the sender of the *SUB* message.

DAT: This control message includes data sent by the producer. Receiver parses the message and retrieves the requested data.

2.3.3 Multimedia content

The main advantage of TRAMP Data Exchanger is that it has been designed to efficiently handle multimedia content [13].

The most important feature is that subscribing to a data segment uses the write-driven strategy. Therefore producers and replicators immediately send data to its subscribers whenever the data segment is changed. That is very important for multimedia applications because of the low latency we achieved by implementing this design.

2.3.4 Architecture of TRAMP Data Exchanger

In this subsection, we analyse design details of *TRAMP Data Exchanger* and describe its implementation.

TRAMP Data Exchanger is a collaboration platform implemented as an user-space daemon in C language. It runs as a stand-alone process and it communicates

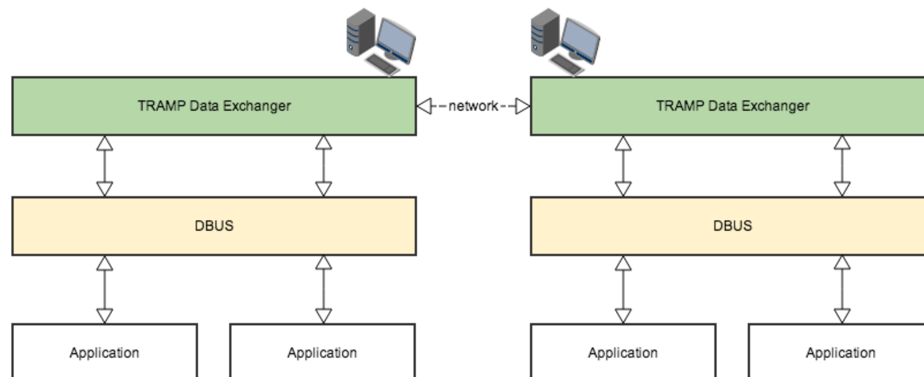


Figure 2.2: TRAMP Data Exchanger architecture overview

with applications using *DBUS Session Bus* as illustrated in Figure 2.2. If a process needs to exchange data with the process located on another machine, this machine needs to run *TRAMP Data Exchanger* daemon. These daemons communicate together over the network.

Figure 2.3 shows that *TRAMP Data Exchanger* initially runs two threads - **DBUS listen thread** and **Server listen thread**.

DBUS listen thread listens to the applications calls from the current machine delivered by a session bus. Application is dynamically linked with *libtramp* the library which provides us with four functions:

- tramp_initialize
- tramp_publish
- tramp_get
- tramp_subscribe

Function *tramp_initialize* allocates shared memory segment and returns a pointer to a corresponding address in a local memory. Other functions are wrappers for sending corresponding messages to the message bus. These functions allow us to instruct *TRAMP Data Exchanger* daemon to run *handle_rpc_publish*, *handle_rpc_get*, *handle_rpc_subscribe* handlers. We describe these calls later in this chapter.

Server listen thread listens on a certain port and connects new instances of *TRAMP Data Exchanger* running on other machines. For each accepted connection, it creates the **connection** thread responsible for communication with a remote daemon.

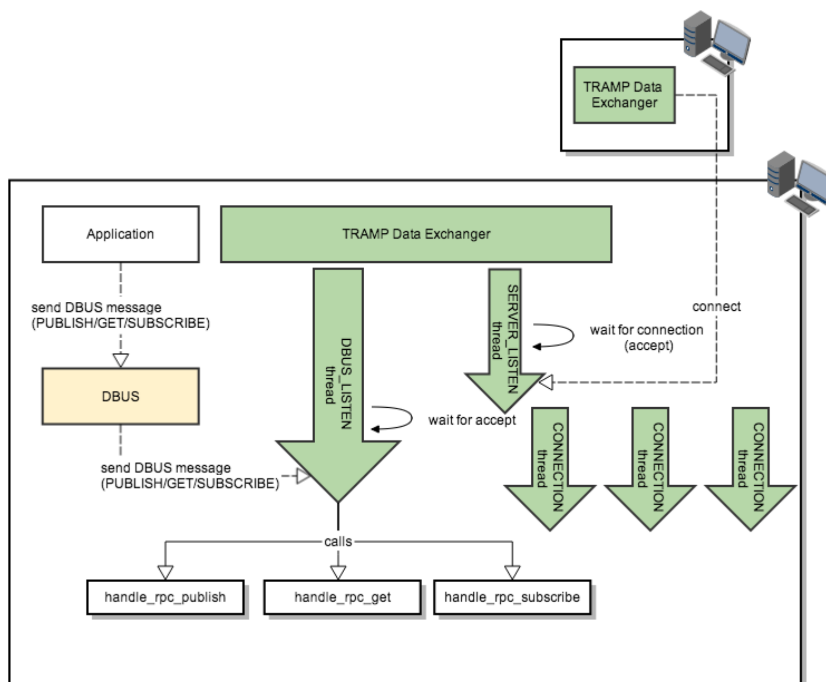


Figure 2.3: Detailed design of TRAMP Data Exchanger

Connection thread

Connection thread is responsible for handling remote messages sent from other *TRAMP Data Exchanger* daemons. Its responsibility is to react to all internal messages described in Subsection 2.3.2 as illustrated in Figure 2.4. It parses the message sent through the network and invokes corresponding functions based on the type of the message. If necessary, it replies with another message to the calling daemon.

Data thread

According to [13], subscribing is used whenever a consumer wants to control when to update the data segment,

"The data propagation with Subscribe uses a write-driven strategy where producing and replicating devices immediately send data to their subscribers whenever new data becomes available" (p. 4).

Data thread is a thread created as a consequence of *subscribe call*. It compares a data segment to the shared memory segment and whenever it notifies a change in the memory, it sends a *DAT* message to the consumer.

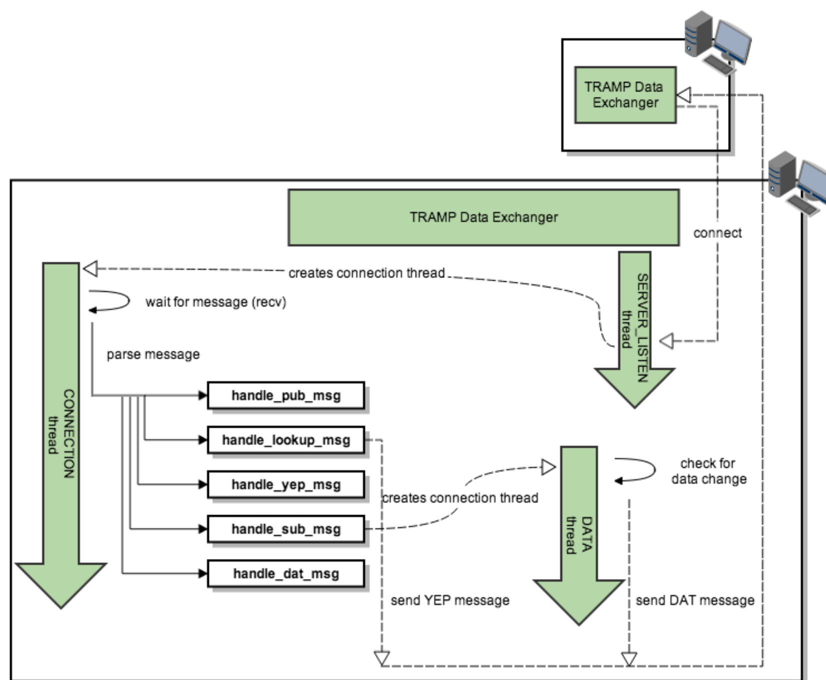


Figure 2.4: Connection thread functionality

Publish/Get/Subscribe RPC handlers

As we can see in Figure 2.3, *PUBLISH*, *GET* and *SUBSCRIBE* functions invoked by an application call *handle_rpc_publish*, *handle_rpc_get*, *handle_rpc_subscribe* handlers respectively. In this subsection, we describe their functionality in more detail.

handle_rpc_publish function creates a *PUB* message and broadcasts it over the network to all connected *TRAMP Data Exchange* daemons. Their *connection* thread parses the message and saves a given label to an internal structure.

handle_rpc_get broadcasts a *LOOKUP* message to all connected daemons. It waits for *YEP* messages from daemons that own the data segment. It is followed by calculating the lowest latency and sending the *GET* message to the node with the lowest latency.

handle_rpc_subscribe behaves in the similar way as *handle_rpc_get*. It also broadcasts a *LOOKUP* message and calculates the lowest latency, but instead of sending a *GET* message to the node with the lowest latency, it sends a *SUB* message, therefore a *data* thread is created in the remote daemon as illustrated in Figure 2.4.

Latency is not calculated every time we want to retrieve a data segment. It is calculated and cached only once.

Internal messages structure

We have already presented the way how *TRAMP Data Exchanger* communicate using internal messages. Now we examine the structure of these internal messages.

PUB	*PUB;label*
LOOKUP	*LOOKUP;label;size*
YEP	*YEP;label;inherited-delay*
GET	*GET;label;size*
SUB	*SUB;label;size*
DAT	*DAT;label;size*

Table 2.3: Internal message structure

The structure is illustrated in Table 2.3. Every message is divided into multiple parts separated by a semi-colon. The first part describes the type of the message. The second part provides the label that the data segment message is used for. Based on the type of the message, additional information can be provided, for instance the size of the data segment or inherent delay.

Once the message is received, **connection** thread parses the message and calls all corresponding functions to handle the message.

2.4 GStreamer

2.4.1 Foundations

GStreamer [31] is pipeline based multimedia framework which helps programmer's to write multimedia applications.

The key idea of GStreamer's design is its pipeline which defines the data flow. A pipeline is formed by various elements, as shown in Figure 2.5. Elements are included in the framework or written by third party programmer's. GStreamer handles management of these elements, data flow and negotiation of the formats. It is not restricted to handle multimedia formats only. It can handle any type of data stream.

2.4.2 Elements

Element is the most important entity in the GStreamer design. Every single *GStreamer* application is formed by a pipeline consisting of GStreamer elements.

By constructing a pipeline, we define the application behaviour. There are three main types of elements in GStreamer - sources, filters and sinks.

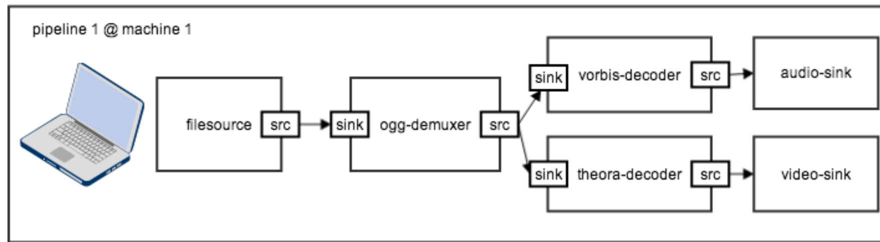


Figure 2.5: Example of GStreamer pipeline

sources

Sources are special types of elements that generate data. The source element is linked as a first element in the pipeline and its role is to fill the pipeline with data. Examples of sources are *audiotestsrc* which produces audio at a certain frequency or *qtkitvideosrc* which provides the pipeline with a video stream recorded using a camera.

filter-like elements

Filter-like elements are elements that receive a data stream, process it and provide it for other elements in the pipeline. This includes filters, codecs, muxers, demuxers and elements for protocol handling. A typical example of a filter is a *videoflip* element which flips and rotates a video. Another is *videobalance* which adjusts contrast, saturation, hue and brightness on a video stream. These types of elements can be linked with multiple elements. A *ogg-demuxer*, for example, is linked with several elements as illustrated in Figure 2.5.

sinks

Sinks are in the role of consumers in the pipeline. They receive data streams and perform an action, usually to output the stream to a sound card, or display the video on a screen. They do not provide data for other elements in a pipeline.

By constructing a pipeline, we define the behaviour of multimedia applications. Suppose we want to write an application which records a video from a web camera and displays it on a screen. An application can be written by constructing a pipeline with two plugins - video recorder plugin and video sink plugin.

In every *GStreamer pipeline*, there is at least one *source* element and at least one *sink* element.

2.4.3 Plugins

Plugins are loadable blocks of code which contain one or more elements. Elements are packed into plugins so we can use them in GStreamer pipelines. Plugins are usually shipped in the form of a dynamically linked library or a shared object file.

2.4.4 Pads

Elements produce and consume data once they are linked in the pipeline. The way data flow is produced or consumed by the plugin is defined using *Pads*. *Pads* are objects associated with elements through which data flows in or out of an element. They handle negotiation of the formats so they can restrict the type of data that flows through the elements. There are two types of pads in GStreamer - source pads and sink pads. They are pictured in Figure 2.5.

2.4.5 Properties

Most GStreamer elements have a set of customizable properties. These properties are useful in two ways. They can be used to determine the elements internal state or to modify the elements behaviour.

Since GStreamer elements are all derived from GObject [27], properties can be set using *g_object_set* and retrieved using *g_object_get* function calls.

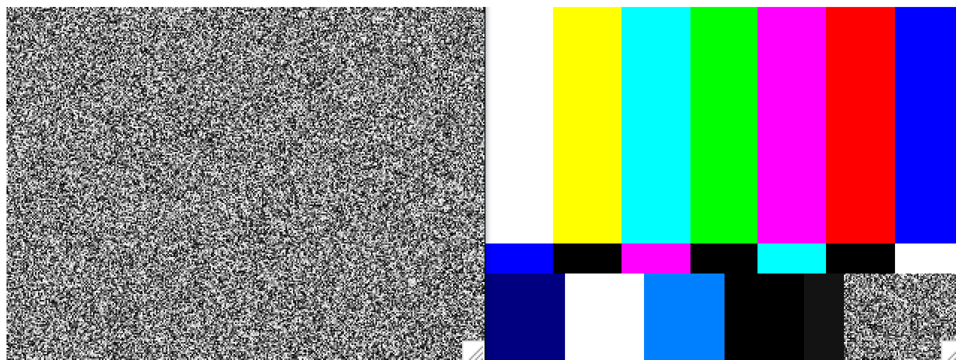


Figure 2.6: Example of GStreamer pipeline with different value of a *videotestsrc pattern* property

To demonstrate properties functionality, let's consider a simple pipeline consisting of two elements. *videotestsrc* and *ximagesink*.

Videotestsrc element creates a simple video stream for testing purposes. *Ximagesink* element renders a video stream in the window on X window system.

Figure 2.6 demonstrates the different behaviour of two pipelines. On the left side of the picture, we have set the *pattern* property of *videotestsrc* element to *snow*, while on the right side the property is set to the default value - *smtpe* which makes an element to produce standard SMTPE test pattern video stream.

2.4.6 Communication

GStreamer provides several mechanisms for communication and data exchange between the application and the pipeline as seen in Figure 2.7.

buffers are objects used to deliver streamed data from one element to another. Since streaming is always done from sources to sinks, these objects always go downstream.

events are objects used for communication between elements or between an application and elements. Since, in some cases, communication needs to be performed both upstream and downstream, events can travel in both directions. In addition, downstream events can also be synchronized with the data flow.

queries are objects used to query an element about specific information. They are similar to *events* described above, but unlike *events*, queries are always synchronously answered. Queries can be used by the elements or the application to query information about the current state of an element or the whole pipeline. They can travel both upstream and downstream. Example of a query is a capabilities query. It returns information about possible capabilities an element can process in its current state. Another example of a query can be a query used to find out the information about the duration of a video stream.

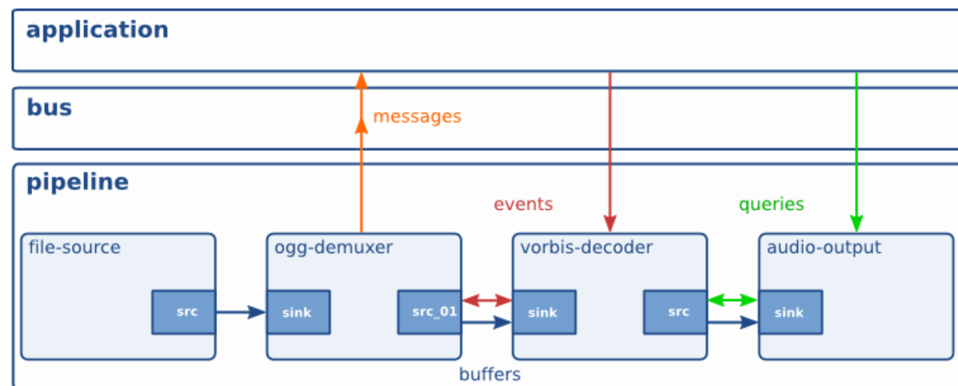


Figure 2.7: Example of a GStreamer pipeline with different communication flows

Source: [30]

2.4.7 Capabilities negotiation types

GStreamer framework is not restricted to handle multimedia only. It can handle any type of data stream. Therefore it is required to have a control over the data that flows through the pipeline. Objects describing the data stream are called capabilities objects.

Capabilities of a pad object specify what kind of data stream can pass through the pad. They are stored in the structure *GstCaps* describing a set of media formats. They can also be expressed as plain text.

Capabilities negotiation is the process of finding media formats between two elements that both elements can handle. There are three types of negotiation in GStreamer.

- Fixed negotiation
- Transfrom negotiation
- Dynamic negotiation

Fixed negotiation

In this type of negotiation, an element can produce one format only. The type of the format is usually hard-coded in the code. Examples of elements producing fixed formats are some of the source elements that can produce one media format only and thus cannot be renegotiated.

Transform negotiation

In this technique, the output format depends on the input format and optionally on the elements properties. Transform negotiation is used by converting elements like encoders and decoders when output format is dependent on input format. Another group of elements where transform negotiation is used is the group of elements that want to operate in a pass-through mode - they don't change input format, but pass it to the next element.

Dynamic negotiation

Dynamic negotiation is the most complex GStreamer negotiation technique. Format is negotiated based on the format the consumer element can handle. The chosen format is usually the one that requires the least amount of effort to produce than the others. Dynamic negotiation is used in some source elements that can produce streams in multiple formats e.g. *videotestsrc* or *v4l2src*.

2.4.8 Capabilities negotiation process

There are two ways that data can flow through the pipeline in GStreamer.

- push mode
- pull mode

In the push mode, an upstream element invokes the pushing of a buffer into the downstream element. In the pull mode, a downstream element asks the upstream element for the data. Each of these modes has its advantages and disadvantages. Discussing the difference between pull mode and push mode is out of the scope of this thesis. For simplicity we focus on a pull mode.

In pull mode, negotiation works as illustrated in Figure 2.8:

1. upstream asks downstream for data it can handle
2. downstream suggest formats it can handle
3. upstream decides on the format it will produce
4. upstream checks if downstream can really handle the chosen format
5. downstream answers that it can handle the chosen format
6. once the format is chosen and downstream can handle it, upstream instruct downstream to reconfigure itself for a chosen format
7. buffers starts to flow through the pipeline

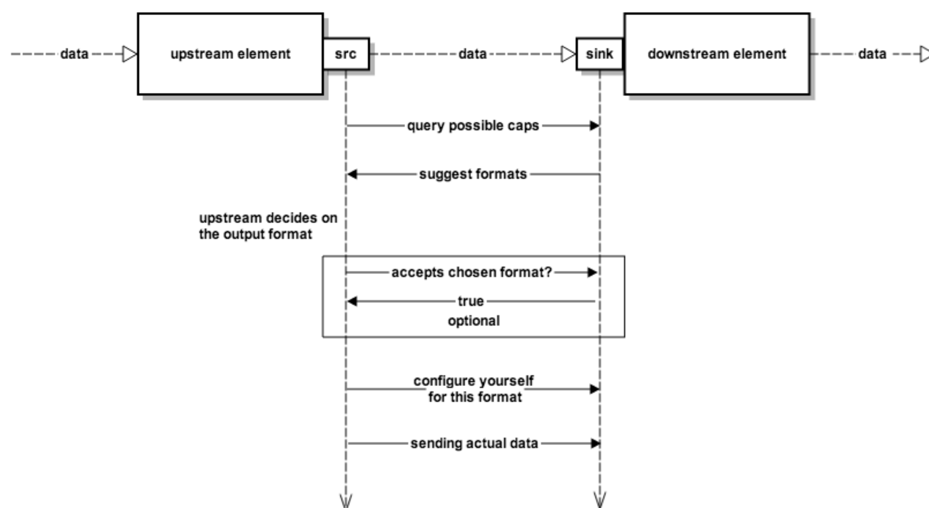


Figure 2.8: Negotiation between two elements

2.4.9 Renegotiation

Let's assume we have a simple video application. The pipeline consists of two elements. The first element records the video stream using a web camera while the second element renders this stream on the screen. In certain circumstances, the user

can decide to resize a window where the video is being rendered. Since the video output itself is not capable of rescaling, video sink needs to ask upstream to change the format. This process is called renegotiation. The process of renegotiation is illustrated in Figure 2.9.

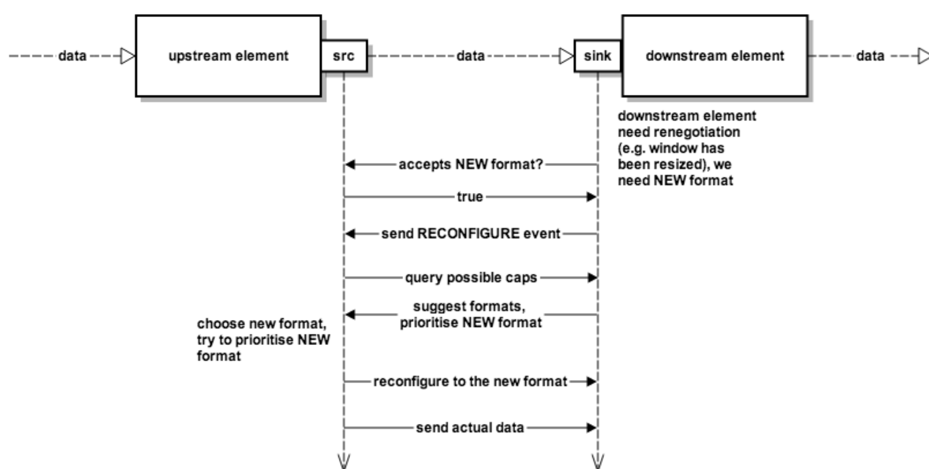


Figure 2.9: Renegotiation between two elements

In the first step, downstream element decides that it wants to receive a different format. Typical use cases include resized video window or some change in a video or audio configuration. Downstream element asks upstream whether it can accept the new format. It continues suggesting new formats until it receives a positive answer. This is followed by sending a special type of event called a *RECONFIGURE* event. As a consequence, upstream element knows that is should repeat the negotiation phase. It repeats all the steps visible in the Figure 2.8. The only difference is that the downstream element now answers differently, prioritizing a new format.

Elements that operate in the fixed negotiation mode drop the *RECONFIGURE* event, because their format is fixed and thus cannot be renegotiated. Elements operating in the transform negotiation mode can handle the *RECONFIGURE* event and recursively send it upstream. Elements using dynamic negotiation first check if they can produce a new format. If so, they reconfigure themselves to handle it. Otherwise, they pass the *RECONFIGURE* event upstream [28].

2.4.10 Structure of GStreamer elements

We have implemented the GStreamer plugin which includes two elements. The following subsection describes the basic structure of sink and source elements in GStreamer from the plugin developer's view. The most important functions described in this subsection are listed in Table 2.4.

plugin_init	called to initialize the plugin
gst_<name>_class_init	one time only initialization of the class
gst_<name>_init	initliaze current instance of the plugin
query	handles queries
event	handles events
render	used by a sink element to output a data buffer
create	used by a source element to insert data to a pipeline
get_property	gets an element's property
set_property	set a property for an element

Table 2.4: List of GStreamer element internal functions

Element metadata

In Subsection 2.4.2 we have described what is a GStreamer element. A GStreamer element defines number of metadata information. These include:

- The name of the element
- The type of the element
- A short description of the element
- Author of the element

This information is statically set during the initialization of the plugin.

Pad template

Subsection 2.4.4 describes the role of *Pads* in the GStreamer framework. Pad template defines properties of pad objects which include the short name of the pad, the type of the pad (downstream/upstream) and its supported capabilities.

Constructing an element

To construct an element, *plugin_init* function is called as soon as the plugin is loaded. This function returns information that is stored in the central registry of the GStreamer framework. It also returns the name, which is used to call two other initialisation functions.

gst_<name>_class_init This function registers the elements metadata and pad templates. It also defines functions that are implemented at a later stage.

gst_<name>_init Initialize function instantiates pads from the elements pad templates. We can also initialize all the important variables in this function.

2.5 VLC media player

VLC is a free open source multimedia framework created by the VideoLAN organization. Its most important feature is modularity. VLC framework is multiplatform and runs on the majority of the most popular operating systems and platforms such as Windows, OS X and Linux. The VLC framework is based on modules that provide frameworks with needed functionality. These modules can be of various types - decoders, muxers, demuxers, filters, etc. Several collaborating modules form an application.

The core VLC framework is used to wire these modules together. Its most important structure is called *libVLCcore*. It manages modules, threads and is responsible for synchronizing audio and video tracks. Modules communicate using a certain interface. They are built against *libVLCcore*.

VLC module In this paragraph, we describe the basic structure of the VLC module. Each VLC module contains two important properties:

- the capability
- the score

The capability is used to determine the category of the module and its functionality. The score is used to set the priority of the plugin within the same class.

For example, when the VLC needs a demuxer, it search for a module with a *demuxer* capability. It prioritises modules with a higher score within this category. It loads them by calling *Open()* function. If it succeeds, it uses a module as a demuxer. If it does not succeed, it tries to open another module with lower score. [1]

In order to choose a module with a specific name, we must instruct the VLC to use its passing modules name. The way how we specify it on the command line is shown in Listing 2.5.

Listing 2.5: Choosing a specific video output in VLC

```
vlc video.avi --vout caca
```

VLC Module types There are several main types of VLC modules. We list the most important ones in table 2.5.

Each of these module types implements a different functionality. In the following paragraphs, we describe each category and its responsibity.

Access modules handle the input and the output. They usually implement access to a device such as a web camera or access data using various protocols e.g. *http*.

Type	Short description
Access	handling I/O
Demux	demuxers
Decoder	decoding a stream
Video filter	video filters
Video output	outputting video
Audio filter	audio filters
Audio output	outputting audio

Table 2.5: VLC module types

Demuxers are modules responsible for extracting the content of a given format. Once VLC opens a file or a stream, it tries to automatically choose the right demuxer.

Decoders are responsible for decoding a stream and passing it to the corresponding output device. Example of a decoder is *ffmpeg video/audio decoder* which is able to decode *.mpg* files.

Video/Audio filters modify decoded video/audio streams. Examples of such functionality can be the rotation of a video stream and various video effects including brightness and contrast adjustment.

Video/Audio output modules pass decoded video/audio to the corresponding hardware such as a video card.

Internal structure of a VLC Module In order to develop a VLC module, we should understand the structure of the code. Following text describes the implementation details of a VLC module.

Listing 2.6 shows the VLC module sample [2].

Listing 2.6: Code structure of VLC module

```
#include <stdlib.h>
/* VLC core API headers */
#include <vlc_common.h>
#include <vlc_plugin.h>
#include <vlc_interface.h>

/* Forward declarations */
static int Open(vlc_object_t *);
static void Close(vlc_object_t *);

/* Module descriptor */
vlc_module_begin()
```

```
    set_shortname(N_("ShortName"))
    set_description(N_("Description"))
    set_capability("access", 0)
    set_callbacks(Open, Close)
    set_category(CAT_INPUT)
    // optionally define an attributes
    add_integer(name, value, text, longtext, advanced)
vlc_module_end ()

/**
 * Starts our example interface.
 */
static int Open(vlc_object_t *obj)
{
    // allocate internal state
    msg_Info(intf, "Hello %s!", who);
    return VLC_SUCCESS;

error:
    // deallocate internal resources
    return VLC_EGENERIC;
}

/**
 * Stops the interface.
 */
static void Close(vlc_object_t *obj)
{
    msg_Info(intf, "Good bye %s!");

    // deallocate internal resources
}
```

First of all, all VLC modules need to identify themselves by setting their description. This description starts with *vlc_module_begin()* and ends with *vlc_module_end()*. All VLC modules need to have their short-name and description defined along with their category, capability and score.

As seen in the code, functions *set_shortname()*, *set_description* sets the module short name and description respectively. These are followed by function *set_capability* which sets the capability of the module to **access**, which means that the module is of access type providing I/O. *set_capability* also sets the score of the module to 0. That means if we need this specific module we need to explicitly specify it using its short name, because of its low priority.

Configuration category is set by *set_category()* function. The VLC framework comes with the integrated user interface that can help us configure the program. Category for instance defines where the module appears in the configuration UI panel.

Each modules behaviour can be modified by setting various properties and parameters. These parameters can be set on the command line or in the user

interface. In order to define the modules parameter, we have to set this parameter in the module definition. This can be done by functions *add_integer*, *add_string*, *add_bool*, etc. as seen in the code sample.

Open function is called by the VLC framework to open a module. This function should initialize all variables and return *VLC_SUCCESS* in case a module can be used. All modules need to have this function implemented.

Close function is another mandatory function required by the VLC framework. It is called once the module needs to be unloaded.

Other functions needs to be specified based on the type of the module. For instance *access* modules need to implement the *seek()*, *control()* and *read()/block()* functions. Modules used for demuxing need to implement *demux()* and *control()* functions. The type of the function which needs to be implemented for a specified module can be found in the documentation for each module type.

Chapter 3

Design

In this chapter, we present the design of our multimedia plugin which is able to handle multimedia content distributed over several machines in a small federation of devices. In Section 3.1 we start by describing our goals with respect to our requirements presented in Section 1.3. Section 3.2 presents reasons for choosing the underlying data propagation framework. We choose a multimedia framework for our plugin in Section 3.3. In Section 3.4, we wire these two frameworks together by introducing the design of our multimedia plugin.

3.1 Goals

This section presents the goals of our design with respect to our requirements and findings in Chapter 2. Our main goal is to make an easy-to-use tool able to manage multimedia content distributed over multiple machines. With this tool the user is able to play synchronized multimedia streams on several machines.

Our first goal is to choose a data propagation framework that fulfils our requirements as well as a multimedia framework that serves as a wrapper for this data propagation tool. The next goal is to integrate these two frameworks together.

The result of our work is a multimedia framework plugin. With our plugin, the knowledge of this framework is sufficient to write distributed multimedia applications.

3.2 Data propagation framework

One of the tasks is to choose a data propagation framework for our plugin, because it is responsible for providing content in a distributed environment. Since our multimedia plugin serves as a wrapper for this tool, it inherits some of its properties like portability and performance.

We compare frameworks introduced in Chapter 2 with respect to the requirements we presented in Chapter 1 focusing on performance and portability. We chose a data propagation framework that fulfils these requirements. Another aspect

mentioned in the requirements is how easy-to-use is the chosen framework. Since our main contribution is making the work with this framework easier for application programmers, this aspect is not as important as for example, performance. In spite of that we analyse how easy-to-use each of these frameworks are, so we can find out whether integration into multimedia frameworks is useful and if it can eliminate the complexity of a DSM platform for application programmers.

Performance *TRAMP Data Exchanger* is optimized to work with multimedia content. It internally builds latency-based distribution trees so we can retrieve data segments from the node with the lowest latency for this data segment. Unlike Linda, its operation does not block. That is very important for multimedia applications because of the possibility of increased latency. Low latency is neither guaranteed in Munin, because of release consistency, which can postpone the propagation of data.

Portability Since *Munin* is implemented as a page fault handler which needs to be registered with the kernel, it can be problematic to use on multiple different platforms and operating systems. *TRAMP Data Exchanger* is implemented as a user space daemon which is more easy and convenient to use in a multiplatform environment. Most of the *Linda* implementations are also multiplatform, so it does not take too much effort to develop and deploy systems on multiple operating systems.

Easy-to-use *TRAMP Data Exchanger* may not be so easy to use for application programmers because of the fact that shared data segments need to be allocated by the framework. Memory allocated by other frameworks and libraries cannot be used by the framework without manually copying memory blocks. We further address this issue in Subsection 3.5.4.

With *Linda*, we need to explicitly put data segments into the tuple space and use the specific API.

The advantage of Munin is that we only need to annotate shared variables, and we can use regular *read/write* operations. Kernel's page fault handler handles underlying data propagation for us.

It is hard to define the term "easy-to-use" in this context. While working with TRAMP and Linda, we need to explicitly call their API, the advantage of Munin is that we can work with the memory using regular library calls. A need to explicitly call the framework's API made us classify TRAMP and Linda as less user friendly than Munin.

Table 3.1 illustrates advantages and disadvantages of these platforms with respect to our requirements.

To decide about data propagation framework, the most important factor for us is performance and ability to run on multiple platforms. From our perspective, it is not

	Munin	Linda	TRAMP Data Exchanger
Good performance	✗	✗	✓
Portability	✗	✓	✓
Easy to use	✓	✗	✗

Table 3.1: Comparison of data sharing platforms

so important how easy-to-use the framework is, because it will be wrapped by our multimedia plugin. Therefore this information is not relevant for the endpoint users.

Based on these facts, we have chosen *TRAMP Data Exchanger* as the data propagation framework for our plugin.

3.3 Multimedia framework

3.3.1 Overview

TRAMP Data Exchanger provides us with the abstraction of linear address space distributed over several nodes and is tailored to the needs of multimedia applications. Linear address space is the natural way for programmers to work with the memory.

We claim that for writing multimedia applications, even higher abstraction is needed. Writing a multimedia application could be easier for application programmers if the framework provides them with more functionality. In the current state TRAMP provides functions to allocate, read and write to the shared memory. There are several multimedia frameworks like *GStreamer* or *VLC* which provides programmers with useful tools e.g. clock synchronization and multimedia format negotiation. They also provides plenty of plugins and modules which includes video/audio filters, codecs, muxers, demuxers etc.

There is a need for integration with some of these multimedia frameworks. Since *TRAMP Data Exchanger* is a software DSM which makes it possible to write multiplatform applications, we should choose a framework which is not tied to one specific platform or operating system. Such a multimedia framework makes programmer's development of multimedia applications easier.

3.3.2 GStreamer

One possible choice is GStreamer. It has been ported to a wide range of processors and operating systems like Microsoft Windows, OS X, Linux or other mobile operating systems including Symbian OS, Android and iOS.

The key advantage of GStreamer is its pipeline based design. As previously described in Chapter 2, GStreamer defines an interface for elements which are connected together and form a pipeline. Because of this well-defined interface, any functionality can be supported in GStreamer by building a customized plugin. These plugins contain elements that are used in the pipeline and cooperate with other elements. Even the functionality of other multimedia frameworks is ported to

GStreamer. An example is GStreamer FFMpeg plug-in, which contains decoders and conversion elements wrapping the functionality of FFMpeg libraries. Once we implement a plugin with support for *TRAMP Data Exchanger*, applications can easily combine functionality of this data sharing platforms with the benefits of GStreamer just by plugging our element into the pipeline.

3.3.3 VLC

Another option is to integrate *TRAMP Data Exchanger* into the VLC framework. VLC also comes with a clean modular design. *TRAMP* can be integrated as a VLC module linked against *libVLCcore*. VLC is multiplatform and supports a wide range of operating systems. One of its advantages is its user interface which provides a friendly way to control media streams.

TRAMP Data Exchanger can be implemented as a *access* module. There are several *access* VLC modules that send streams using HTTP protocol. Our VLC plugin would send data using *TRAMP Data Exchanger* instead.

3.3.4 Choosing the multimedia framework

Both *GStreamer* and *VLC* frameworks provide us with the functionality needed for writing multimedia applications. They both have a modular design that allows us to write an extension. While *GStreamer* provides us with its pipeline and possibility to write customized elements, VLC provides an opportunity to write a module to integrate *TRAMP*.

Both frameworks are widely used and in both cases, customized user interfaces can be easily built to interact with the media in a more easier way.

One of the most important reasons to integrate *TRAMP* into multimedia frameworks is to verify that it can be used by multimedia applications. After trying both frameworks, we have chosen *GStreamer* for *TRAMP Data Exchange* integration, because it provides us with better tools for testing the daemon. Using its pipeline design, we can easily modify an application on the command line as illustrated in Listing A.1.

3.4 Detailed design

In the previous section, we have chosen GStreamer as a multimedia framework that we develop a plugin for, and also *TRAMP Data Exchanger* as our data propagation framework. This section presents design details of our multimedia plugin and presents important *GStreamer* features that are useful for interaction with *TRAMP Data Exchanger*. It mostly focuses on how these two frameworks interact with each other.

Figure 3.1 illustrates the usage of our plugin and the most important aspects of our design. In this figure, our plugin is used to build an application which plays a video file located on the *Machine 1*. The application has three components

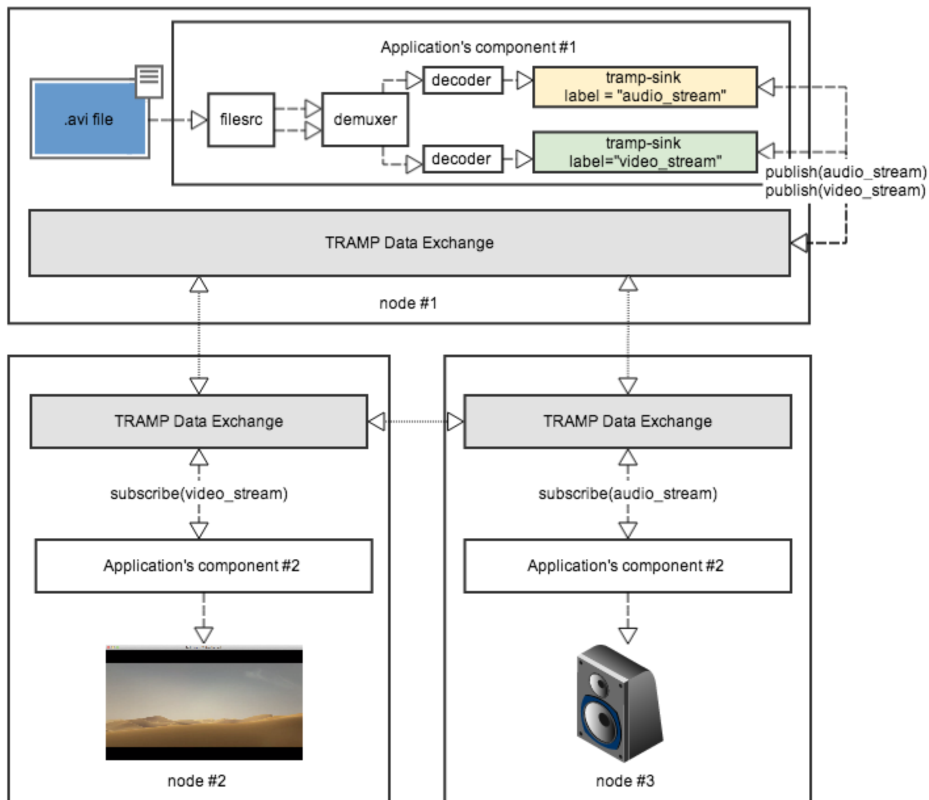


Figure 3.1: Architecture of our system

distributed over three different machines. *Component #1* is used to read a source video file, *Component #2* renders video frames and *Component #3* takes care of handling audio output.

3.4.1 TRAMP plugin

Support for additional features in GStreamer is implemented in the form of elements. Elements can then be plugged into the pipeline and extend the pipeline's functionality. These elements are packed together as a package. For instance previously mentioned GStreamer FFMpeg plugin contains several GStreamer elements including decoders, demuxers, elements for colour-space conversion and others.

The output of our work is a GStreamer plugin – a loadable block of code containing two elements - **GstTrampSink** and **GstTrampSrc**. This GStreamer plugin is shipped in the form of a dynamically linked library. *GstTrampSink* is a sink element consuming data in one pipeline and making it accessible for several *GstTrampSrc* source elements, which act as producers in other pipelines.

3.4.2 Architecture overview

In order to send multimedia content using our plugin, *TRAMP* daemon needs to be running on each device and GStreamer framework needs to be installed. The application can communicate with *TRAMP* daemon directly using its API, use the GStreamer plugin or both. Architecture overview is illustrated in Figure 3.2. Our plugin has two dependencies - *TRAMP Data Exchanger* libraries and GStreamer.

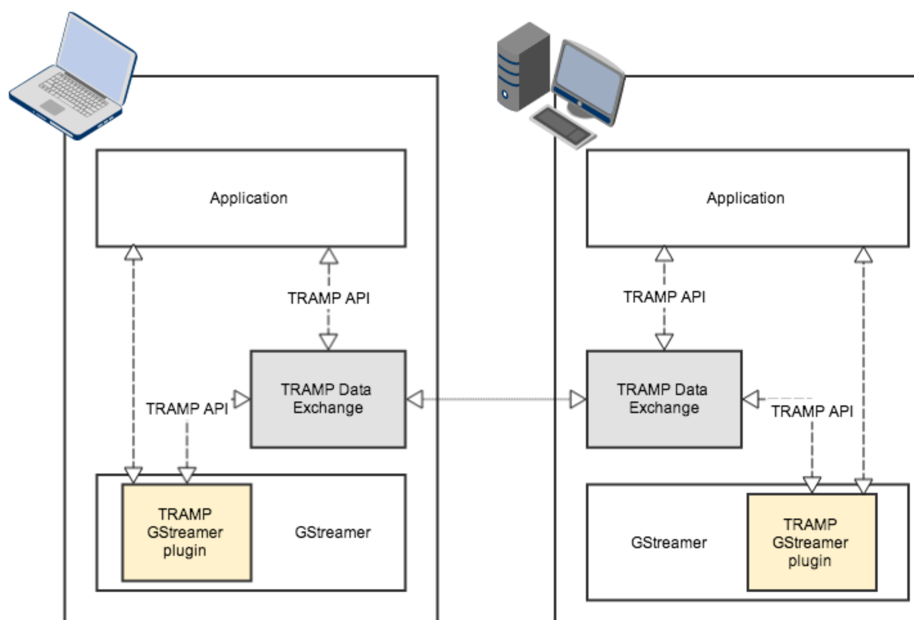


Figure 3.2: Architecture overview

3.4.3 TRAMP GStreamer elements

GStreamer is a powerful framework and if we want to have a pipeline operating on several devices, there are several plugins we can use. An UDP plugin containing UdpSink/UdpSource elements [1] is able to transmit streams over the network from one device to another one.

By using UpdSink/UdpSource elements, we can send packets from one pipeline to another pipeline possibly located on different machines using UDP protocol. What we want instead is to support features of *TRAMP Data Exchanger* and take advantage of its features e.g. building underlying latency-based distribution trees.

With GStreamer plugin supporting *TRAMP Data Exchanger* functionality, application programmers do not need to allocate data segments and take care of handling data streams. All they need to do is to link *GstTrampSink* at the end of the pipeline

[1] http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good-plugins/html/gst-plugins-good-plugins-udpsink.html

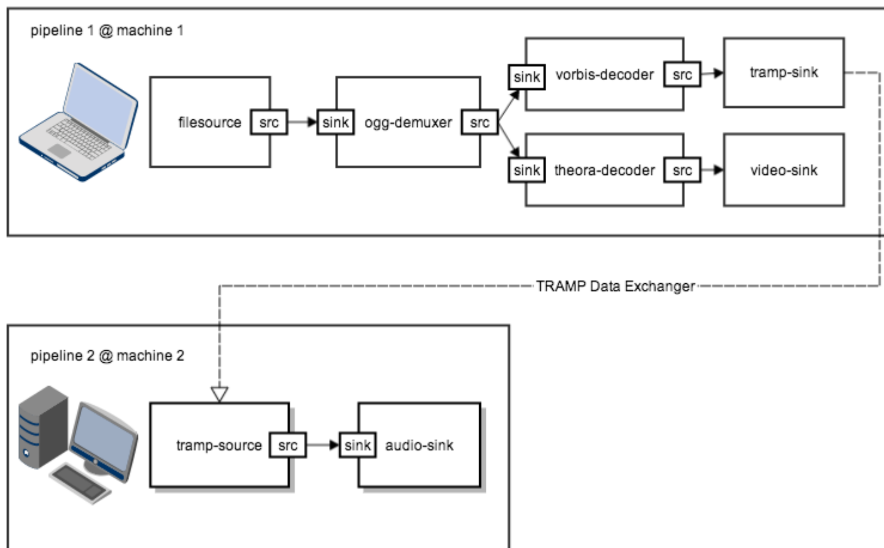


Figure 3.3: Example of two GStreamer pipelines interacting using TRAMP daemon

and link *GstTrampSrc* at the beginning of the other pipeline. That means programmers are able to work with the daemon in the same way as they are used when using the UDP plugin.

Data received in the sink is available in the source element on the other pipeline. The plugin and runtime system takes care of building underlying distribution trees and data propagation. The player illustrated in Figure 2.5 taking advantage of GStreamer plugin with *TRAMP Data Exchanger support* is shown in Figure 3.3

3.4.4 Labels

As shown in Figure 3.3, *GstTrampSink* and *GstTrampSrc* elements are not necessarily part of the same pipeline. Thus, we cannot use standard GStreamer linking mechanisms mentioned in Section 2.4.4.

The problem was already present in the case of the UDP GStreamer plugin. Authors of *GstUdpSink* and *GstUdpSrc* the elements solved this problem by setting properties on the GStreamer elements, discussed in Section 2.4.5 In this case, a programmer can set host and port properties and thus define an address where the UDP elements communicate.

Since we implement the GStreamer wrapper for the TRAMP Data Exchanger, we can take advantage of labels associated with data segments we discussed in Section 2.3.1. We use the GStreamer properties to pass the element an information about the label of the data segment it should work with.

An example of code used by application programmer is shown in Listing 3.1.

Listing 3.1: Setting a property on the sink element

```
g_object_set (gst_tramp_audiosink, "label", "audio_stream", NULL);
```

Figure 3.4 illustrates how *TRAMP* runtime system can be instructed to use different data segments using GStreamer labels.

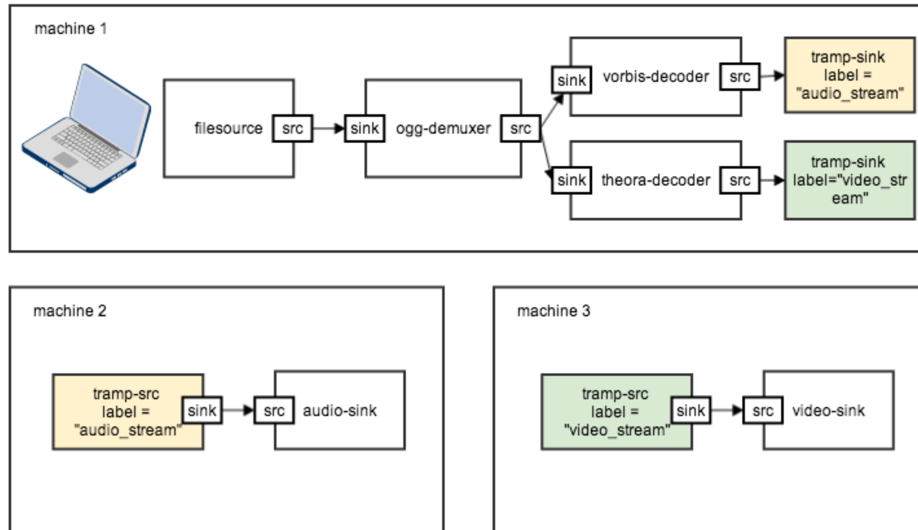


Figure 3.4: Using GStreamer labels with TRAMP Data Exchanger

3.4.5 Caps negotiation

In this subsection, we discuss the type of negotiation technique we use in our plugin. We need to choose one negotiation technique from those mentioned in Subsection 2.4.7. Our GStreamer plugin is not bound to any specific format and supports any type of data stream. Therefore the chosen negotiation technique should not restrict it to any specific type of data and should not affect the pipeline behaviour. Applications using our plugin should run in the same way as if they ran on the same machine without using a DSM system.

In the following paragraphs we discuss possible advantages and disadvantages of using various negotiation techniques by our plugin.

Fixed negotiation does not fulfill the requirement that our plugin should handle any type of data stream. If we use a fixed negotiation, the type of data stream would be hard-coded into the plugin. Our plugin could support, for instance, only specific types of audio streams.

Dynamic negotiation is the most flexible negotiation technique. Our element operates in the pass-through mode, it does not modify the stream. Therefore it does not have to modify the capabilities of the stream. It passes the same data through the DSM system. Dynamic negotiation depends on downstream capabilities, it usually does a *CAPS* query to find the appropriate transform. This is not needed by our plugin and it would increase the complexity of our code.

Transform negotiation is the best fit for our plugin. Our GStreamer TRAMP plugin operates in the pass-through mode, because we do not want to change the format of the data, but transfer it from one pipeline to another. The type of media format received on the sink pad of the *GstTrampSink* is proxied to the *GstTrampSrc* element. In case renegotiation is needed, we proxy all required queries and events in both directions (upstream and downstream).

3.4.6 Communication between pipelines

Our GStreamer plugin virtually connects two separate GStreamer pipelines. In order to achieve this, the pipeline needs to be able to access any information about the data from the other pipeline. The basic principles of GStreamer's communication mechanisms are described in Subsection 2.4.6.

We introduce the problem using the situation illustrated in Figure 3.4. In this example the application is composed of three pipelines distributed over three different machines. The stream source is the file located on *Machine 1*. The application decodes this file and sends the audio stream to the *Machine 2* and the video stream to the *Machine 3* where these streams are processed.

Once *tramp-sink* element is linked in the pipeline on the first machine, it receives information about the capabilities of the stream in order to identify what kind of data it will handle. Since it distributes this stream to *Machine 2*, it needs to share this information with the *tramp-src* element located in the other pipeline on *Machine 2*.

Another use case where information needs to be shared among pipelines can be also found in Figure 3.4. Let's assume that *video-sink* located on *Machine 3* renders the video stream using X Window System. Once the user decides to resize this window, the *video-sink* element receives a *RECONFIGURE* event which instructs it to change the format of the video stream. It needs to pass this event upstream to the producer of data - *tramp-sink* located on *Machine 1*.

In this chapter, we discuss how we can pass information from one pipeline to another. The most important metadata we need to send is the capabilities information, so the downstream pipeline can configure itself to receive this data. We also need to pass information about different events and queries our elements receive.

There are many ways to pass the capabilities information from *GstElementSink* to *GstElementSrc*. We discuss three of them.

1. Using additional data segment

2. Piggybacking

3. Proxying events & queries

Additional data segment can be shared using the *TRAMP Data Exchanger*. We could define a convention that each shared data segment has its capabilities stored in the data segment published with label prefixed with *_tramp_capabilities_*.

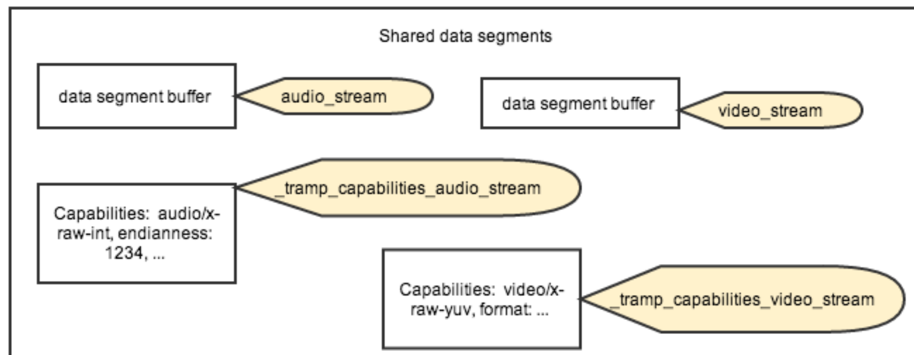


Figure 3.5: Using additional shared data segment to share the capabilities information

If *GstTrampSink* and *GstTrampSrc* have their label property set to *video_stream*. *GstTrampSink* publishes the capabilities information in the data segment with the label *_tramp_capabilities_video_stream* from where it can be retrieved by the *GstTrampSrc* element with the *TRAMP daemon's get* call. This situation is illustrated in Figure 3.5.

We have chosen this prefix because of the convention that functions prefixed with underscore are in general considered to be private (e.g. in Python) or not recommended (man 2 _exit).

Piggybacking information into the shared buffer is another possible implementation of how to send information about the capabilities.

Data buffer containing shared multimedia data could also include information about the capabilities. There are several advantages of this approach. The most important one is that the capabilities in GStreamer can dynamically change and be renegotiated. Therefore it should be possible for the element to determine what kind of data is in the specific buffer. Another advantage of this approach is its simplicity, we set information about the capabilities to the location after data block as illustrated in Figure 3.6.

Proxying events & queries is the most convenient way that information about the capabilities can be sent from one pipeline to another. Since our elements operate in the pass-through mode, their responsibility is to proxy relevant queries and

Size	Field
1 byte	seq_no
4 bytes	buffer_size
buffer_size bytes	buffer_data
string terminated by \0	capabilities (data type in the buffer)

Figure 3.6: Internal buffer used by our plugin

events. The most important queries and events we need to support are illustrated in Figure 2.9.

Once the pipelines are connected together, the source element in the downstream pipeline sends the *RECONFIGURE* event to the upstream pipeline which is then proxied upstream to the source of the data. All of the relevant queries and events that follow are proxied as illustrated in Figure 3.7.

Another advantage of this solution is that once we implement support for proxying queries & events, any two elements in two connected pipelines can communicate using whatever query they need. We list some of the important queries that are useful to proxy in Table 3.2 and Table 3.3.

Name	Description
End of Stream	Sent when the stream is finished
Capabilities	Sent to instruct downstream to reconfigure to given format
Reconfigure	Sent to upstream to start renegotiation
Gap	Sent to downstream if there is no data for a certain amount of time
Quality Of Service	Event containing information about the real-time performance of the stream

Table 3.2: Important events to proxy

We implement two versions of our plugin. The first simple version implements communication between our elements using additional data segment. The second, more complex solution implements the proxy for events and queries. Even though

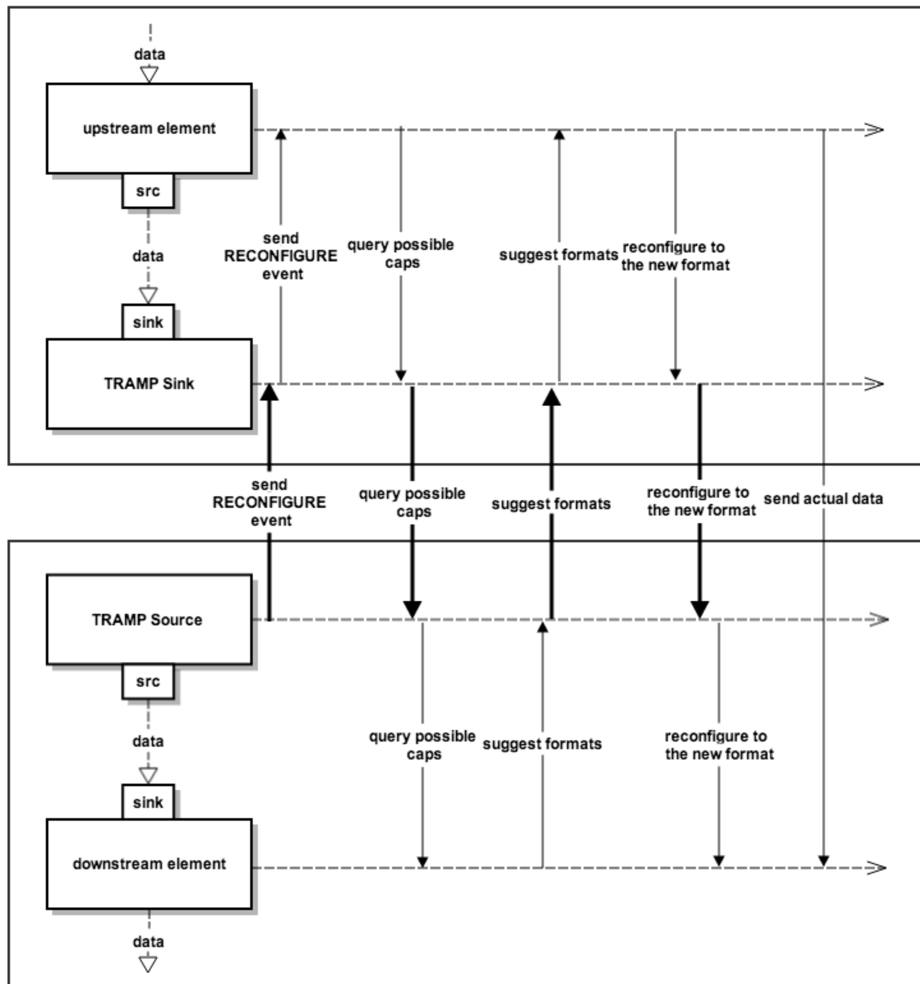


Figure 3.7: Proxying events & queries

the second solution is more generic, we have not managed to fully implement it because of several unresolved issues of the *TRAMP Data Exchanger*. We discuss these issues later in this chapter. Therefore implementation of the first simple version allows us to improve the TRAMP exchanger platform and evaluate our design.

In the following paragraphs, we focus on the design of the second, more complex version. We discuss the renegotiation process which is possible in case a proxy for events and queries is implemented.

Renegotiation Let's assume we have a simple video application. The video source is located on Node A, while the video window displaying the output is located on node B. The user controlling node B decides that he wants to change the dimensions

Name	Description
Position	Query the current position of playback
Duration	Query the total length of the stream
Capabilities	Query the possible capabilities that downstream element can accept in the current state
Accept caps	Query to find out whether specific capabilities are accepted by an element

Table 3.3: Important queries to proxy

of this window. Since the video output itself is not capable of rescaling, video sink needs to ask upstream to change the format. This is handled by the renegotiation process mentioned in Subsection 2.4.9.

This perfectly works within one pipeline. What we need to guarantee is that everything works in the same way when using our plugin.

As seen in Figure 2.8 and Figure 2.9, elements linked together exchange data about the format with two important queries - *ACCEPT* and *CAPS* queries. The solution to our problem is to proxy these queries.

The upstream element also sends *CAPS event* to instruct the downstream to reconfigure itself for given capabilities object. In Subsection 3.4.5, we claim that our elements use transform negotiation. Therefore we also need to pass *RECONFIGURE* event to the upstream element. Our plugin thus needs to proxy *RECONFIGURE* and *CAPS* events as well.

3.4.7 Proxy implementation

In the previous Subsection 3.4.6, we have decided to proxy all relevant queries and events. In this subsection, we explain how this proxy is designed in more detail.

Type	Name	Description
unsigned char	counter	The message number
enum QUERY, EVENT, REPLY	type	type of the message
int	metadata	Metadata specific for each type
char[SIZE]	data buffer	Data buffer containing event/query data

Table 3.4: Structure t_message

We have created a data structure *t_message* to achieve this. This data structure is described in Table 3.4. Both sink and source elements publish this data structure and listen to each other as illustrated in Figure 3.8.

Every time an element updates its buffer, it increases the counter variable by one. Any other element that is subscribed to this buffer periodically checks this variable. Once it detects that the variable is incremented, the subscriber can read the contents of the buffer and wait for another change again. The element that reads

the buffer then identifies the type of the message based on the *type* variable. It can represent query, event or reply to the query.

Each of these queries and events have specific types as listed in Table 3.2 and Table 3.3. The type of query/event is stored in *metadata* variable. Data buffer contains additional parameters for the query/event, for instance capabilities string.

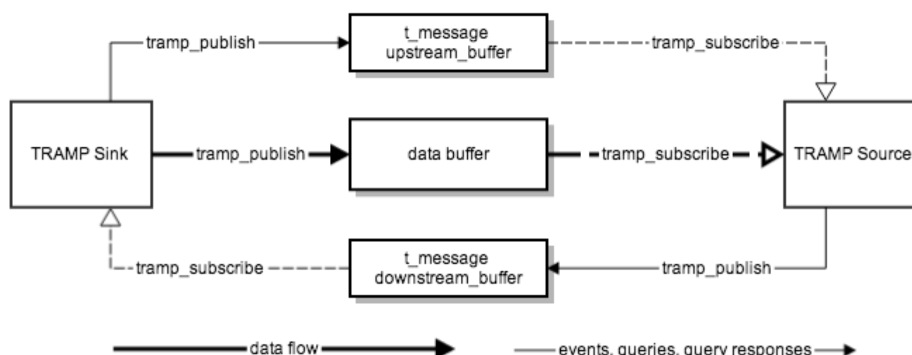


Figure 3.8: Proxy implementation

Figure 3.8 shows that we use a separate buffer for sending data and a separate buffer for sending events and queries from upstream. Since both of these buffers deliver data in the same direction, they could be merged into one buffer. To make the design more clear, we have decided to use two separate buffers.

Let's give an example of sending a *capabilities* query from upstream. Since we want to send query downstream. We change the *upstream_buffer*, because that's what the downstream element is subscribed to.

Name	Value
counter	counter + 1
type	QUERY
metadata	GST_QUERY_CAPS
data buffer	video/x-raw, width=(int)624, height=(int)352, ...

Table 3.5: Sending capabilities query downstream

We set the type to *QUERY*, set metadata to *GST_QUERY_CAPS* and copy the string representation of the capabilities to the data buffer. Once we are done we increment the counter variable by one, letting downstream read our message. We illustrate the contents of the buffer published by the upstream element in Table 3.5.

3.4.8 Data replication

One of the key features of *TRAMP Data Exchanger* which helps us to decrease latency is replication of data and building distribution trees based on latency. In

other words, if we subscribe to a given data segment, we subscribe to an owner of this data segment with the lowest latency possible.

Component can own data segment in two cases.

1. It is the producer of a given data segment

2. It is subscribed to a given data segment, thus it always owns the latest updated version of this data

As shown in Figure 3.9 and Figure 3.10, if *Node #2* subscribes for data segment *audio_stream*, it can replicate it for *Node #3*. *Node #3* can thus receive this data segment with the lowest latency.

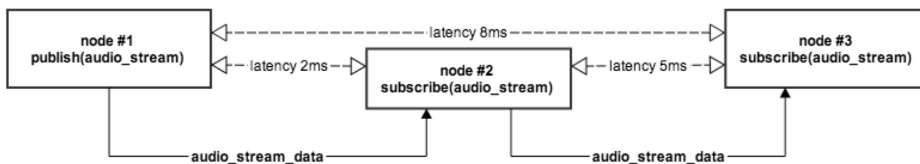


Figure 3.9: TRAMP: Example of data propagation with replication

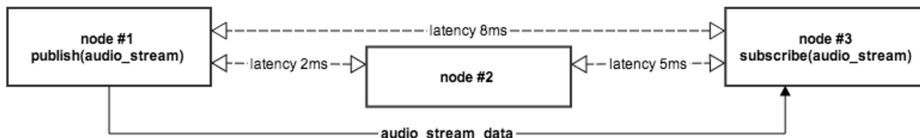


Figure 3.10: TRAMP: Example of data propagation without replication

3.4.9 Using multiple TRAMP Source elements

This approach works if data from our *TRAMP Sink* element is sent to at most one *TRAMP Source* element. If we want to send data to multiple *TRAMP Source* elements, we would need to duplicate the stream and create a new *TRAMP Sink* element. That would not make much sense since we would publish the data stream with a different label and we would lose the valuable feature of the *TRAMP Data Exchanger* - replication of data and building latency-based distribution trees.

In the simple implementation of our plugin, *GstSrcElement* reads the upstream data without sending any type of messages to the upstream, so multiple elements of this type can subscribe to one *GstSinkElement* without any problems. In the second, more complex implementation, communication between several elements needs to be coordinated in a more complex way. Designing such a solution is left for future work.

3.5 Analysing TRAMP Data Exchanger

As we already mentioned in the Chapter 1, the features of the DSM system we chose are very important for the final performance of our plugin. We have described *TRAMP Data Exchanger*'s design and implementation in Section 2.3. In this section, we point out some of the issues we found during the development of the multimedia plugin. After presenting an issue, we offer several possible solutions how the problem can be fixed. One of the goals of our thesis is to verify that TRAMP can be efficiently used in the multimedia applications domain. Identifying important issues that prevent us to do so is therefore very important. Another reason to list and describe these issues is that some of them need to be necessarily fixed in order to implement the complex version of our plugin with the support for renegotiation and communication between elements in both directions.

3.5.1 Updating peers with inconsistent data

As illustrated in Figure 2.4, on each *SUB* message a daemon accepts, additional **data thread** is created which is responsible for pushing data to the subscribers immediately after they change.

Algorithm 1 Data thread pseudocode

1: $shm \leftarrow shared_memory_segment(label)$
2: $reply_content \leftarrow NULL$
3: **while** $True$ **do**
4:     **if** $shm \neq reply_content$ **then**
5:         $reply_content \leftarrow shm$
6:         $reply \leftarrow new_dat_msg(reply_content)$
7:         SEND($reply$)
8:     **end if**
9: **end while**

Algorithm 1 presents the pseudocode of the *data* thread shown in Figure 2.4. It is executed in the context of the *TRAMP Data Exchanger* daemon.

The variable *SHM* is a reference to the shared memory segment. This memory segment is shared with the application process in Algorithm 2. Data thread compares this data segment with the part of the memory that was last sent to the subscriber with *memcmp* [21] as presented in Algorithm 1 line 4.

Consumers usually subscribe to data segments because they want to receive the latest consistent version. For example, in the context of multimedia applications, they want to receive the latest video frame generated by the producer. In case the producer is not completely ready to copy the latest video frame to the shared data segment, *memcmp* notifies the change in the part of the shared memory segment and sends incomplete, corrupted data to the consumer.

Algorithm 2 Example of multimedia application

1: $shm \leftarrow$ tramp_initialize($label$)
2: tramp_publish($label$)
3: ...
4: $shm \leftarrow new_video_buffer()$
5: ...

Considering that Algorithm 1 runs as part of the data thread in data sharing daemon and Algorithm 2 runs as a standalone process, a problem can occur if Algorithm 2 is interrupted in the middle of processing on line 4. Meanwhile Algorithm 1 compares the memory segment and sees that it has changed, which leads to sending corrupted data to the subscriber. We need to somehow prevent the daemon from doing this. Following paragraphs discuss possible ways for achieving this.

Notifying daemon about memory change is one possible way we can prevent this problem from happening. The producer would not create a *data* thread for each subscriber, but instead an additional API function *tramp_push* would be called from the multimedia application to instruct the daemon that the memory has changed. Replicators would push an updated data segment on arrival of the *DAT* message.

However, this approach has several weaknesses. First of all, *tramp_push* would need to block until a daemon is ready with the propagation of the data to the consumer. Otherwise the application could modify the data segment before a daemon completes this task. Since communication is done through the message bus, it would need to wait for the DBUS message confirming the successful delivery from the daemon. Otherwise, a daemon could create a temporal copy of this data and process it asynchronously. If not necessary, we want to avoid copying memory blocks because of increased latency.

Another drawback about this solution is that the application programmer would need to explicitly notify the daemon about the changed data segment.

Mutual exclusion Another possible way of doing this is mutual exclusion. Data exchange daemon runs concurrently with the application and communicates using the system message bus. Therefore using synchronization primitives can solve this issue. We must make sure that the memory segment is changed atomically and that the data sharing daemon does not propagate changes of the data segment until the application is done modifying it.

Even though we want to avoid blocking in our application, this mutual exclusion is needed and never blocks for a long period of time. We have considered using named semaphores or file locks. We prefer to use file locks to named semaphores, mostly because they are auto-released in case the application crashes or receives a signal. It is true that a signal can be caught and resource can be released in the

signal handler, however some signals like *SIGKILL* [19] and *SIGSTOP* [19] cannot be caught or ignored.

Algorithm 3 Data thread pseudocode with file locking support

1: $lockfd \leftarrow$ open(FLOCK_NAME)
2: $shm \leftarrow shared_memory_segment(label)$
3: $reply_content \leftarrow NULL$
4: **while** $True$ **do**
5:     LOCK($lockfd$)
6:     **if** $shm \neq reply_content$ **then**
7:         $reply_content \leftarrow shm$
8:         $reply \leftarrow new_dat_msg(reply_content)$
9:         SEND($reply$)
10:     **end if**
11:     UNLOCK($lockfd$)
12: **end while**
13: close($lockfd$)

Algorithm 4 Example of multimedia application with file locking support

1: $lockfd \leftarrow$ open(FLOCK_NAME)
2: $shm \leftarrow$ tramp_initialize($label$)
3: tramp_publish($label$)
4: ...
5: LOCK($lockfd$)
6: $shm \leftarrow new_video_buffer()$
7: UNLOCK($lockfd$)
8: ...
9: close($lockfd$)

Algorithms 3 and 4 show a way how to put file locks in the code to prevent race conditions. This solution requires an application programmer to use file locks explicitly in the code of the application. Instead we decided to add three additional TRAMP API calls which wrap this code. They are listed in Table 3.6.

3.5.2 Rebalancing distribution trees

TRAMP Data Exchanger is optimized for multimedia applications. When the application needs a data segment, it broadcasts a *LOOKUP* message and then waits for any *YEP* messages from those who own it. The *YEP* message contains the information about the delay from the original producer. The application then chooses the data segment with the lowest latency.

Latency in the network can change, for example because of the churn, mobility or an unstable network. The *TRAMP Data Exchanger* does not react to latency

int tramp_lock_init (tramp_lock_t *lock, const char* label)	initialize the lock for given label
int tramp_lock(tramp_lock_t *lock)	locks a data segment
int tramp_unlock(tramp_lock_t *lock)	unlocks a data segment

Table 3.6: TRAMP API extension

changes in the network. From [13] *"Changes in network conditions may require subsequent re-organization of the distribution tree. Re-balancing the distribution tree in such events is left for future work." (p. 3).*

Solution

TRAMP Data Exchanger stores the label for each data segment it is subscribed to, together with the opened connection to the provider of this data segment. The provider is the replicator with the lowest latency from the original source. A possible solution is to continuously broadcast a *LOOKUP* message. A *LOOKUP* message could be broadcasted in one of the following ways.

1. An additional thread can be created which broadcasts *LOOKUP* messages in certain time intervals, with the exception of the original producer

2. An *LOOKUP* message can be broadcast on every *n-th* access for current data segments.

We claim that the first option has better performance. Broadcasting on every *n-th* access can keep the latency outdated for rarely accessed data segments.

3.5.3 Subscribing to the data segment that is not yet published

TRAMP allows only one producer per data segment as stated in [13], *"We allow only one producer per data segment as an alternative to locks. This avoids blocking, but application developers need to be aware of this in advance and implement their components accordingly, e.g., by adding a dynamic suffix to the label." (p.3).*

It is up to the application to make sure that any data segment is published before anyone can subscribe to it. Figure 3.8 shows the scenario where no such behaviour is possible. For instance if two applications want to subscribe to each other's data segment at start-up, the application which starts first is not able to locate a data segment of the other application. In the current state *TRAMP* cannot handle such a situation.

Solution

There are several possible solutions for this problem:

1. While trying to subscribe, we can continuously broadcast *LOOKUP* message until we succeed
2. Implement error handling in *TRAMP Data Exchanger*, right now all the library functions return *void* pointer, they could return an error code on failure
3. Once re-balancing of distribution trees is implemented, this problem disappears, because we continuously send *LOOKUP* messages in certain time intervals, after we subscribe. Once the producer application publishes its data segment, *LOOKUP* message succeeds and we receive *YEP* reply.

We claim that the combination of a second and third solution is the best one. *TRAMP Data Exchanger* misses system for error handling which would allow applications to decide how they want to react to certain situations. The first solution could block the application for a very long time or forever in the worst case.

3.5.4 Copying memory to the shared data segment

Once data segment is requested from *TRAMP*, it gives us a pointer to a shared memory segment. We can work with this chunk of memory as it is regular memory allocated by the *malloc* [20] call. We can read and modify this piece of memory and changes are propagated to the other peers by the *TRAMP* framework.

In case we allocate memory in our application on our own, we can just replace *malloc*-like calls with *tramp_initialize*. The problem occurs when a data segment we work with is allocated by a library and we cannot change the way how the data is allocated. If this data needs to be shared with other nodes, we need to copy the data between these two locations, for instance using *memcpy* [22]. Coping memory in this way is expensive, time consuming and can cause increased latency in a multimedia application.

In spite of this fact, in the current implementation of our plugin we copy memory allocated by GStreamer to the shared memory allocated by the TRAMP framework. We are unable to find an alternative solution to deal with this problem.

Chapter 4

Implementation

In this chapter, we describe the implementation of our GStreamer plugin. In Section 4.1 we present an overview of our implementation. Section 4.2 describes design details of the elements included in our plugin.

4.1 Overview

We have implemented our project as a GStreamer plugin in C programming language. It is shipped in a form of a plugin represented as dynamically linked library. This GStreamer plugin contains two elements - *trampsink* and *trampsrc*. It is tested on Linux and OS X platforms. A version for Windows and mobile platforms is possible but not yet implemented. In order to use our plugin in the GStreamer pipeline, programmer uses shared libraries with sink and source elements.

We have implemented two versions of our project in two separate branches.

Stable branch In the stable branch, we implemented a simple version of our plugin. The main purpose of this implementation is to verify the correctness of our design and demonstrate that with TRAMP, the multimedia content can be shared between several nodes while fulfilling our requirements.

In this version, the elements use the transform negotiation technique without the support for events/queries proxy described in Subsection 3.4.7. We use additional data segments to provide downstream with the capabilities of the stream. In this version, multiple consumers can subscribe to get data segments from one producer element. That means that data is published with the same label, which allows a distribution tree to be built based on the latency in the network.

The name of the stable branch is *origin/master* in our Git repository. The access to our Git repository is described in Appendix B.

Development branch The development branch is the experimental branch which is not fully implemented yet. This complex version of our plugin uses transform negotiation and supports the proxy for GStreamer events and queries. Therefore,

instead of publishing capabilities information using additional data segments as we do in the stable branch, capabilities are negotiated thanks to the events/queries proxy. This version supports the renegotiation process as well.

When working with pipelines on the same machine, the plugin in the development branch works as anticipated. In the situation where pipelines are located on different nodes, TRAMP Data Exchanger needs to be used to propagate shared data segments to the consumers. Unresolved issues discussed in Section 3.5 cause that this branch is not fully functional in its current state. We have therefore used the stable branch to evaluate our design.

The name of the development branch is *origin/devel* in our Git repository.

4.2 Implementation details

In this section, we present implementation details of the stable branch. This section describes how the most important aspects of our design are implemented in C. The complete implementation of our development branch contains some additional functionality. It can be reviewed from our Git repository referenced in Appendix B.

In order to describe the implementation of our plugin in a detailed way, we describe the three most relevant files we have written. Files *gsttrampsink.c* and *gsttrampsrc.c* implement sink and source elements respectively. The header file *gsttrampcommon.h* defines shared macros and data structures used in both of the afore mentioned files. We separately describe the design of sink and source elements.

Shared resources Listing 4.1 describes the file named *gsttrampcommon.h* which defines macros and the types used in our implementation. *GST_TRAMP_MAXBUFFERSIZE* defines the maximum size of the buffer we share using the TRAMP platform. *GST_TRAMP_MAXCAPSSIZE* defines the maximum size the buffer we use to provide downstream with the capabilities information. *GST_TRAMP_LABEL* is a tramp label we use to share data segments with TRAMP.

Listing 4.1: Defining constants and types

```c
#define GST_TRAMP_MAXBUFFERSIZE 55000
#define GST_TRAMP_MAXCAPSSIZE 1000
#define GST_TRAMP_LABEL "AUDIO_LABEL"
#define GST_TRAMP_CAPS_LABEL "CAPS"

typedef struct t_message {
    unsigned char seq_no;
    uint32_t size;
    char data[GST_TRAMP_MAXBUFFERSIZE];
} t_message;
```

Structure *t_message* is the C analogy of the buffer that is shared using TRAMP which is illustrated in our design in Figure 3.6. The *seq_no* variable is the number

representing the ID of the message. It is incremented by an upstream when a new message is ready to be read by the downstream element. Variable *size* represents the size of the GStreamer buffer we send using this data structure. Pointer *data* is the pointer to the actual data we send.

Sink element implementation In Subsection 2.4.10, we have described the basic structure of every GStreamer element. In the following paragraphs, we present the implementation of relevant functions from Table 2.4.

plugin_init & gst_trampsink_class_init functions are called in order to initialize function pointers, meta data and register our plugin with GStreamer. We set the meta data, pads and initialize function pointers to **set_caps** and **render** functions which are used to send the capabilities and data to the downstream element. We describe the implementation of these functions in Listings 4.4 and 4.9.

Listing 4.2: Initializing meta data and function pointers

```
static void
gst_trampsink_class_init (GstTrampsinkClass * klass)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
  GstBaseSinkClass *base_sink_class = GST_BASE_SINK_CLASS (klass);

  gst_element_class_add_pad_template (GST_ELEMENT_CLASS(klass),
      gst_static_pad_template_get (&gst_trampsink_sink_template));

  gst_element_class_set_static_metadata (GST_ELEMENT_CLASS(klass),
      "GStreamer␣TRAMP␣Sink␣element", "GStreamer␣TRAMP␣plugin", "
          GStreamer␣support␣for␣TRAMP",
      "Jan␣Vorcak␣<janvor@ifi.uio.no>");

  base_sink_class->set_caps = GST_DEBUG_FUNCPTR(
      gst_trampsink_set_caps);
  base_sink_class->render = GST_DEBUG_FUNCPTR(gst_trampsink_render
      );
}

static gboolean
plugin_init (GstPlugin * plugin)
{
  return gst_element_register (plugin, "trampsink", GST_RANK_NONE,
      GST_TYPE_TRAMPSINK);
}
```

gst_trampsink_init function initialize a new instance of a sink element. As shown in Listing 4.3, we first initialize a sink pad. In the second step, we allocate new

shared memory segments for data and capabilities and publish it to all connected devices. As a last step, we initialize lock in order to avoid consistency issues mentioned in Subsection 3.5.1.

Listing 4.3: Initialization of a new sink element instance

```
static void
gst_trampsink_init (GstTrampsink *trampsink)
{
  // initialize sink pad
  trampsink->sinkpad = gst_pad_new_from_static_template (&
      gst_trampsink_sink_template
      , "sink");

  // initialize and publish data buffer
  trampsink->seq_no = 0;
  trampsink->msg = (t_message *) tramp_initialize(GST_TRAMP_LABEL,
       sizeof(t_message));
  bzero(trampsink->msg->data, GST_TRAMP_MAXBUFFERSIZE);
  tramp_publish(GST_TRAMP_LABEL, sizeof(t_message));

  // initialize and publish capabilities buffer
  trampsink->caps = (char *) tramp_initialize(GST_TRAMP_CAPS_LABEL
      , GST_TRAMP_MAXCAPSSIZE);
  bzero(trampsink->caps, GST_TRAMP_MAXCAPSSIZE);
  tramp_publish(GST_TRAMP_CAPS_LABEL, GST_TRAMP_MAXCAPSSIZE);

  // initialize the lock
  trampsink->lock = tramp_lock_init(GST_TRAMP_LABEL);
}
```

gst_trampsink_set_caps function is used to send the capabilities to the downstream element. Our sink element receives them in a form of **GstCaps* caps** pointer from the upstream element. Our implementation copies this information to the published memory segment so that the downstream element can retrieve them with **tramp_get** API call.

Listing 4.4: Implementation of gst_trampsink_set_caps function

```
static gboolean
gst_trampsink_set_caps (GstBaseSink * sink, GstCaps * caps)
{
  GstTrampsink *trampsink = GST_TRAMPSINK (sink);
  GST_DEBUG_OBJECT (trampsink, "set_caps");
  strcpy(trampsink->caps, gst_caps_to_string(caps));
  return TRUE;}
```

gst_trampsink_render function is the most important function in our implementation, because it actually sends data downstream. It is invoked by the GStreamer framework when new data is received from the upstream e.g. video source element.

It first locks the shared data segment, so that TRAMP Data Exchanger daemon does not propagate data to other peers until we have finished updating it. It reads the GStreamer buffer and copies its contents to the shared memory. By incrementing a sequence number, we let the source element know that new data is available.

Listing 4.5: Implementation of gst_trampsink_render function

```
static GstFlowReturn
gst_trampsink_render (GstBaseSink * sink, GstBuffer * buffer)
{
  GstTrampsink *trampsink = GST_TRAMPSINK (sink);
  GST_DEBUG_OBJECT (trampsink, "render");

  tramp_lock(trampsink->lock);
  // copy sequence number
  trampsink->msg->seq_no = trampsink->seq_no;
  trampsink->seq_no++;

  // map the buffer
  GstMapInfo info;
  gst_buffer_map(buffer, &info, GST_MAP_READ);

  // copy buffer size to the shared memory segment
  trampsink->msg->size = info.size;

  // copy actual buffer content
  memcpy(trampsink->msg->data, info.data, info.size);

  // unmap
  gst_buffer_unmap (buffer, &info);
  tramp_unlock(trampsink->lock);

  return GST_FLOW_OK;
  }
```

Source element implementation In the following paragraphs, we show how source element handles data in our plugin.

plugin_init & gst_trampsrc_class_init functions functions are implemented in a similar way as *plugin_init* & *gst_trampsink_class_init* functions in the sink element. The only difference is that we set pointers to different functions - **get_caps** and **create**, which are supposed to retrieve data from the upstream.

Listing 4.6: Implementation of gst_trampsrc_render function

```
static void
gst_trampsrc_class_init (GstTrampsrcClass * klass)
{
  GObjectClass *gobject_class = G_OBJECT_CLASS (klass);
  GstBaseSrcClass *base_src_class = GST_BASE_SRC_CLASS (klass);

  gst_element_class_add_pad_template (GST_ELEMENT_CLASS(klass),
      gst_static_pad_template_get (&gst_trampsrc_src_template));

  gst_element_class_set_static_metadata (GST_ELEMENT_CLASS(klass),
      "GStreamer␣TRAMP␣Source␣element", "GStreamer␣TRAMP␣plugin",
          "GStreamer␣support␣for␣TRAMP",
      "Jan␣Vorcak␣<janvor@ifi.uio.no>");

  base_src_class->get_caps = GST_DEBUG_FUNCPTR (
      gst_trampsrc_get_caps);
  base_src_class->create = GST_DEBUG_FUNCPTR (gst_trampsrc_create)
      ;

}

static gboolean
plugin_init (GstPlugin * plugin)
{
  return gst_element_register (plugin, "trampsrc", GST_RANK_NONE,
      GST_TYPE_TRAMPSRC);
}
```

gst_trampsrc_init function initializes a source pad as well as shared data segments for the data and the capabilities.

Listing 4.7: Implementation of gst_trampsink_render function

```
static void
gst_trampsrc_init (GstTrampsrc *trampsrc)
{
  trampsrc->srcpad = gst_pad_new_from_static_template (&
      gst_trampsrc_src_template
      , "src");
  gst_pad_use_fixed_caps(trampsrc->srcpad);

  trampsrc->msg = (t_message *) tramp_initialize(GST_TRAMP_LABEL,
      \
          sizeof(t_message));
  trampsrc->init = 0;
  tramp_subscribe(GST_TRAMP_LABEL, sizeof(t_message));

  trampsrc->caps = (char*) tramp_initialize(GST_TRAMP_CAPS_LABEL,
      GST_TRAMP_MAXCAPSSIZE);
```

```
  trampsrc->lock = tramp_lock_init(GST_TRAMP_LABEL);
  start = time(NULL);
}
```

gst_trampsrc_get_caps function is used to retrieve the capabilities data published by sink's *set_caps* function, so that our source element can inform other elements what kind of data it provides. This is done by reading shared data segments using *tramp_get* API call.

Listing 4.8: Implementation of gst_trampsink_render function

```
static GstCaps *
gst_trampsrc_get_caps (GstBaseSrc * src, GstCaps * filter)
{
  GstTrampsrc *trampsrc = GST_TRAMPSRC (src);
  GST_DEBUG_OBJECT (trampsrc, "get_caps");

  tramp_get(GST_TRAMP_CAPS_LABEL, GST_TRAMP_MAXCAPSSIZE);

  return gst_caps_from_string(trampsrc->caps);
}
```

gst_trampsrc_create function is the function that provides the downstream elements with data. The main purpose of this function is to wait for a memory change in a shared data segment. Once a memory change is detected, it reads data from the shared data segment, sends it to the allocated GStreamer buffer and provides this buffer to the downstream element.

Listing 4.9: Implementation of gst_trampsink_render function

```
static GstFlowReturn
gst_trampsrc_create (GstBaseSrc * src, guint64 offset, guint size,
    GstBuffer ** buf)
{
  GstTrampsrc *trampsrc = GST_TRAMPSRC (src);

  GST_DEBUG_OBJECT (trampsrc, "create");

  if(!trampsrc->init) {
      g_print("Initialized\n");
      trampsrc->counter = trampsrc->msg->seq_no;
      trampsrc->init = 1;
  }

  while(trampsrc->msg->seq_no == trampsrc->counter);
```

```
  tramp_lock(trampsrc->lock);
  uint32_t buffer_size = trampsrc->msg->size;

  *buf = gst_buffer_new();
  GstMemory *memory = gst_allocator_alloc(NULL, buffer_size, NULL)
      ;

  gst_buffer_append_memory(*buf, memory);

  GstMapInfo info;
  gst_buffer_map(*buf, &info, GST_MAP_WRITE);
  memcpy(info.data, trampsrc->msg->data, buffer_size);

  unsigned char miss = trampsrc->msg->seq_no - trampsrc->counter;

  trampsrc->counter = trampsrc->msg->seq_no;
  gst_buffer_unmap (*buf, &info);
  tramp_unlock(trampsrc->lock);

  return GST_FLOW_OK;
}
```

Chapter 5

Evaluation

In this chapter we evaluate the performance and usage of our multimedia plugin. In Section 5.1 we present the basic goals of our evaluation. Section 5.2 discusses different evaluation techniques that are used to evaluate technical aspects. Various factors that affect the results are presented in Section 5.3. We describe the chosen metrics and setups we used to evaluate these metrics in Section 5.4. We present our results in Section 5.5 which is followed by a discussion at the end of this chapter.

5.1 Evaluation goals

The main goals for our evaluation are to verify that our multimedia plugin can be used for real-time multimedia applications and evaluate this plugin in different scenarios. We evaluate the performance and usability of our multimedia plugin with respect to the requirements presented in Section 1.3.

- Goal 1: We check that our multimedia plugin can be used for multimedia applications. The minimal requirements are to keep the latency overhead under 100ms and support the bit rate of minimum 3.75Mbit/s in order to be able to transmit MPEG-2 encoded video stream [13]. We observe and analyse important metrics such as latency, bandwidth, CPU and memory usage which influence the performance of our multimedia plugin.

- Goal 2: To show that our framework is not dependent on a specific platform we set several pipelines running on different operating systems to work together.

- Goal 3: In order to demonstrate that our plugin makes it easier for application programmer to write distributed multimedia applications, we construct several GStreamer pipelines using GStreamer command line utilities. By doing this, we show that integration into a multimedia framework makes the work with TRAMP Data Exchanger easier, because changing the behavior of an

application is as easy as modifying a pipeline - either by adding/removing elements in the pipeline or modifying their properties.

- Goal 4: Compare the performance of our plugin to the GStreamer's UDP plugin. Our plugin has the similar role as an already existing GStreamer UDP plugin, but instead of using an UDP protorol, we take advantage of the TRAMP data sharing platform. Therefore, comparing these two plugins can reveal the scenarios where using TRAMP platform is more efficient than using UDP protocol.

5.2 Evaluation approach

When working on an evaluation, there are two key steps that need to be performed - selecting an evaluation technique and selecting evaluation metrics. There are three main evaluation techniques that can be chosen - *analytical modelling*, *simulation* and *measurement* [14].

Analytical modelling is a theoretical technique which presents a mathematical model and calculates performance for a solution. Simulation is a way to test a solution in a simulated environment which can often be controlled during the process of an evaluation. The third option are measurements performed on a real implementation. The first two approaches can be used mostly for simple projects, because the model or the simulation of a complex solution can also become complex and error-prone.

Each of these techniques has its advantages and disadvantages and can be chosen based on multiple factors. According to [14] there are seven key criteria when choosing an evaluation technique: stage of the project, time it takes to perform an evaluation, availability of tools, accuracy, cost, saleability and the probability of random events influencing the results. In the following paragraphs, we describe two that we consider to be the most important ones for our evaluation.

Stage of the project is an important criterion when choosing an evaluation approach. While analytical modelling and simulation can be used in all of the stages, measurement can only be performed once we have implemented at least a functional prototype.

Accuracy is another important factor. When evaluating a complex system, an analytical model needs to be simplified, which often leads to inaccurate results. Simulations can be more detailed than analytical models and their results are often more accurate. The most accurate results are achieved while measuring an implemented solution or its prototype, even though the environment where measurements run can also differ from reality.

Describing all the factors for choosing the right technique is out of the scope of this paper. [14] describes all of them in more detail. Since we have implemented a

working prototype of a multimedia plugin for the TRAMP data sharing platform, we have decided to use measurement as an evaluation technique. It allows us to test this system on real hardware and get the most accurate results.

5.3 Evaluation factors

When it comes to evaluating the performance and usage of the GStreamer TRAMP plugin, there are several aspects which influence the behaviour of the system. These include the hardware specifications of the collaborating nodes, network speed and the maximum size of the shared data segments configured in the *TRAMP Data Exchanger*. In this section, we describe these factors and their influence on the results.

5.3.1 Node specifications

The specifications of nodes can affect the behavior of our plugin. We have used two types of nodes with various specifications in order to find out if there is any influence on processing power and the results. We present the variety of used hardware in Table 5.1.

	Desktop computer	Laptop
Operating system	Fedora release 20 (Heisenbug)	OS X Mavericks (Version 10.9.2)
CPU	Intel(R) Core(TM) i7 2.93GHz	Intel(R) Core(TM) i5 2.5 GHz
Memory	4 GB 1333 MHz DDR3	4 GB 1600 MHz DDR3
Total Number of Cores	4	2

Table 5.1: Node specifications

Our main goal is to measure the influence of used hardware on the number of processed buffers in certain time intervals. We want to find out in what way computational resources affects the number of buffers TRAMP can process for a given period of time.

5.3.2 Network speed

In our evaluation, network speed has an impact on the latency and the quality of the transmitted multimedia content. The main purpose of using TRAMP for multimedia is transmitting real-time data with minimal latency. We want to find out in what way a network speed influences the results and what kind of multimedia streams can be shared using network interfaces with different capacity.

By testing our plugin in the network with different latency between each pair of nodes, we want to demonstrate that when using our plugin, TRAMP builds distribution trees based on the latency in the network.

5.3.3 Maximum buffer size of TRAMP Data Exchanger

Another important aspect is the configuration of the *TRAMP Data Exchanger* and the maximum size of the shared data segment. As we have already described in Chapter 2, a publisher of the data segment publishes the data segment of a certain size. All the changes to this buffer are then sent over reliable TCP connections if needed. In the situation where the shared data segment is smaller than the maximum allowed size, the message is padded with zero bytes to the maximum size of the buffer. Therefore performance is not dependent on the size of the actual data we send, but on the maximal size of the buffer.

We evaluate the impact of this variable on the overall performance of the plugin and inspect what kind of data streams we can send using TRAMP.

5.4 Metrics

Every element in GStreamer influences the overall performance of the whole pipeline. Therefore, we must focus on the following objectives during the development.

- We want to make sure that our elements can handle as many data streams as possible in order not to restrict a pipeline to a specific format.
- We want to handle the data as efficiently as possible and to avoid adding any additional latency for the data flowing through the pipeline.

We evaluate the GStreamer plugin, which serves as a wrapper for TRAMP. Therefore bad performance can be either caused by our plugin or the *TRAMP Data Exchanger*. We measure important performance metrics of our plugin and identify the possible sources of bad performance. These include:

- Latency
- Processed buffers per time interval
- CPU load
- Memory load

In order to evaluate different metrics, we need to use different scenarios. After introducing a metric we describe and illustrate different setups used to measure it.

5.4.1 Latency

The latency in GStreamer is the time it takes for a data buffer to travel from the source element to the sink element.

We distinguish two types of latency in GStreamer - internal and external latency.

Internal latency is introduced by buffer processing by each element in the pipeline. External latency is influenced by a network delay or delay caused by various hardware.

Our plugin does not include any element that processes the stream. It provides sink and source elements instead. These two elements need to make sure they communicate as fast as possible and in the most efficient way.

Increased internal latency can be caused by an element performing unnecessary computations, repeating the same computation, synchronously performing a task that could be done asynchronously, etc.

Increased external latency is caused by a network delay or hardware.

When optimizing a GStreamer plugin, we want to focus on decreasing internal latency. In our case-study, we examine latency as the time difference between a buffer entering a sink element and the same buffer leaving the source element in the other pipeline. However this latency includes the network delay and the time it takes for a processor to do a context switch.

In general there are two ways we can measure latency:

Measuring the Round-Trip time is the technique we send a buffer to a different pipeline which immediately sends it back. We measure the time it takes for a buffer to return and calculate the latency. A setup for measuring such a task is illustrated in Figure 5.1.

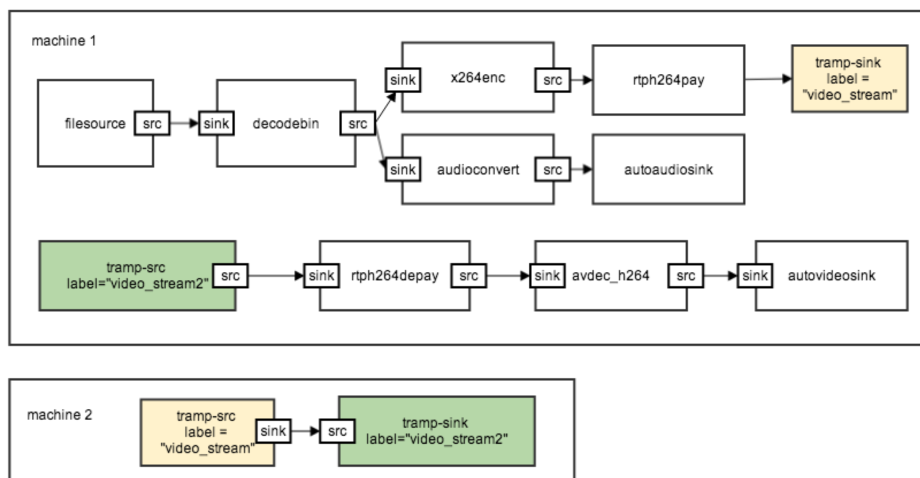


Figure 5.1: Measuring the Round trip time

Synchronizing clock on two machines is another possibility. Once we are sure that clocks on both machines are synchronized, we can send the buffer and measure the time difference. The advantage of this approach is its simplicity. The disadvantage is that if the synchronization between the two machines is not precise, it affects the results of the measurement.

Since Network Time Protocol can be used to synchronize two machines with very high precision [26], we decided to measure latency using the second approach. By synchronizing the clocks before each experiment, we make sure that the time difference is within the milliseconds limitations. Therefore it is acceptable for our results. We illustrate this setup in Figure 5.2.

Evaluation setup

In the first scenario, we measure the latency which is introduced by the network along with the latency introduced by our plugin and TRAMP. As we already described above, there are two possible setups to evaluate the latency - measuring the round trip time and synchronizing clocks on two different machines.

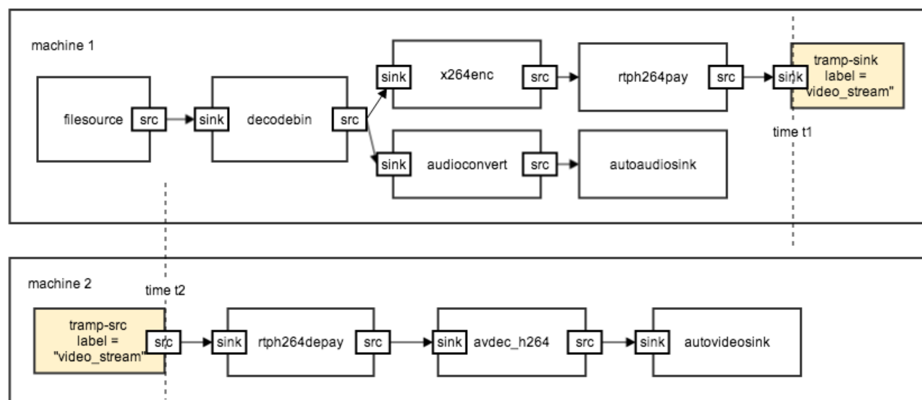


Figure 5.2: Measuring latency on two machines with synchronized clocks

We have decided to measure latency using the setup in Figure 5.2. We have created a setup with two connected desktop computers. The latency between these two computers is approximately 0.5 milliseconds. We have measured the average latency between these two machines using the *ping* tool. It is important to understand that the *ping* tool operates on the network layer. Therefore it does not include the time it takes for a scheduler to perform a context switch or additional overhead caused by a transport layer.

We measure the latency by subtracting the time when the buffer appeared in the sink pad of the *TRAMP source* element *(time t1)* to the time when the very same buffer appeared in the source pad of the *TRAMP sink* element *(time t2)*. If we subtract the real network latency from this number, we get the additional delay

introduced by our plugin including the overhead on the transport layer and the time it takes to do a possible context switch.

In the setup we have chosen, we open a file, encode it using *H264* encoder and packetize it using the *rtph264pay* plugin. The *Rtph264pay* plugin encodes the stream into RTP packets. As we have already stated in our requirements, the internal latency should not exceed 100 milliseconds.

5.4.2 Processed buffers per time interval

This metric shows how many buffers can be processed by our plugin during a certain time interval. The *TRAMP Data Exchanger* works in a way that a producer publishes a data segment while the subscriber is trying to detect any change in the memory and processes the memory in case it has changed.

The problem occurs when the consumer does not have enough time to process every single buffer and misses some buffers while processing the others. The number of missed buffers is dependent on the update frequency of the shared buffer.

Let's consider a situation where we want to send 100MB of data from one machine to another. If we use a 1MB buffer, we need to send one hundred buffers of size 1MB. It takes some time for 1MB of buffer to be copied to the shared memory buffer. Consumers thus have enough time to detect any change in the memory. Copying the memory needs to be done because of the design aspect mentioned in Subsection 3.5.4.

In the case where we slice the same 100MB to 10^8 buffers of size 1 byte, a problem can occur since the consumer is not guaranteed to detect every single memory change. Since it tries to detect the memory change using an infinite loop, it can for instance, end up waiting in the ready queue of the scheduler while the shared memory segment is changed.

In our scenario, we evaluate this metric by sending H264 encoded video packetized by the *rtph264pay* plugin. We change the MTU property of this plugin causing different frequencies of updating shared memory segments. By logging the incoming buffers on the consumer side, we can compare the number of received and expected buffers for a certain video sample.

Evaluation setup

While developing and evaluating our elements, we have found out that the *TRAMP Data Exchanger* is limited by the number of buffers it can handle per certain time interval. These problems occur even in case two pipelines run on the same machine, so we can argue that this issue is not caused by a slow network. However it can be influenced by the performance of the node.

We reuse the scenario from Figure 5.2. The only difference from Figure 5.2 is that in this case two pipelines run on the same node. We have decided to reuse this scenario because of the presence of *rtph264pay* and *rtph264depay* elements. These elements have a very useful *MTU* property. By modifying this property, we

can set the maximal size of the buffer to be sent from this element. By controlling this property, we can increase or decrease the frequency of updating shared memory segment by *TRAMP*.

Every buffer we send has a unique ID which constantly increases. The consuming element can therefore identify how many buffers it managed to process and report the number of missing buffers per certain time interval.

In our test, we send 280 seconds of H264 encoded video stream with different MTU sizes - 26 bytes, 1400 bytes (default value of *rtph264pay*), 6000 bytes and 10000 bytes.

Since we want to identify the source of this problem, we inspect the influence of hardware performance as well as the maximum size of shared buffer configured in the *TRAMP*. Therefore we execute this experiment on two different machines described in Table 5.1.

TRAMP should be able to process the MTU of lower sizes which corresponds to the high update frequency of the shared buffer.

5.4.3 CPU load

Increased CPU load can be caused by the following:

- Unnecessary computation
- Synchronous computation of possibly asynchronous task
- Inefficient code/algorithm
- Unnecessary manipulation with the memory - allocating unnecessary memory blocks, unnecessary *memcpy*, etc.

We measure CPU load using the *top* [18] utility available on UNIX platforms. The *top* utility is an important tool for identifying potential problems with CPU load. It periodically displays CPU statistics of all running processes. It is a tool used for debugging and system administration. Apart from other statistics, the *top* measures and reports the CPU load of the process. A single thread can achieve 100% of CPU load. Multithreaded applications can therefore achieve more than 100% of CPU load. In order to visualize the load of the CPU, we use the *gnome-system-monitor* tool. This tool also visualizes important metrics like memory usage, swap history and network bandwidth in both directions.

5.4.4 Memory load

Measuring memory load can identify several problems in the application's code. The most common problems we can identify are as follows:

- Memory leaks causing increasing memory consumption by a process
- Allocating unnecessary memory blocks

- Unnecessary *memcpy* calls
- Inefficient reusing of the memory

We have concern about these issues because they can increase the latency of the multimedia application, but they can also make the application unstable. The most important tools for measuring memory load are *top*, *gnome-system-monitor* and widely used *Valgrind* [25]. *Valgrind* is a tool which is commonly used to identify existing memory leaks in the application. It can also be used as a memory and cache profiler.

Evaluation setup

We measure the CPU and memory load of our plugin in the same way using two different scenarios. In the first scenario we measure the CPU/Memory load of the sink element. In the second scenario we inspect the CPU/Memory load of the source element.

We have set up two machines running Fedora described in Table 5.1 and we share the audio stream using our plugin. The setup is illustrated in Figure 5.3. For simplicity, we don't illustrate the pipeline anymore. The type of the multimedia stream does not influence the load of the CPU/Memory, since our plugin treats all the streams in the same way.

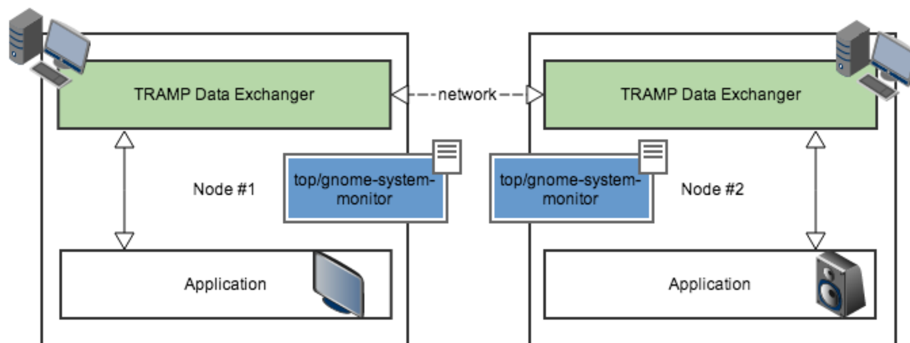


Figure 5.3: Setup for measuring CPU/Memory load

In order to measure the CPU/Memory load of our element, we measure the CPU/Memory load of the same pipeline with and without our element. We compare the load of these two pipelines in order to measure the overhead added by our plugin.

Another important experiment is to measure possible memory leaks of our plugin. We do that by running the *valgrind* tool as well as checking the memory usage using the *top* tool for a longer period of time. Possible memory leaks could make our application crash or increase the internal latency in case we don't properly reuse allocated space.

5.4.5 Comparison with the UDP plugin

GStreamer provides a *gst-good-plugins* package which includes the *UDP Plugin* we already mentioned in the Chapter 2. The *UDP Plugin* also connects several different pipelines possibly running on different nodes. It uses UDP protocol to broadcast the multimedia. The goal of this experiment is to compare the performance of our plugin to the *UDP plugin*.

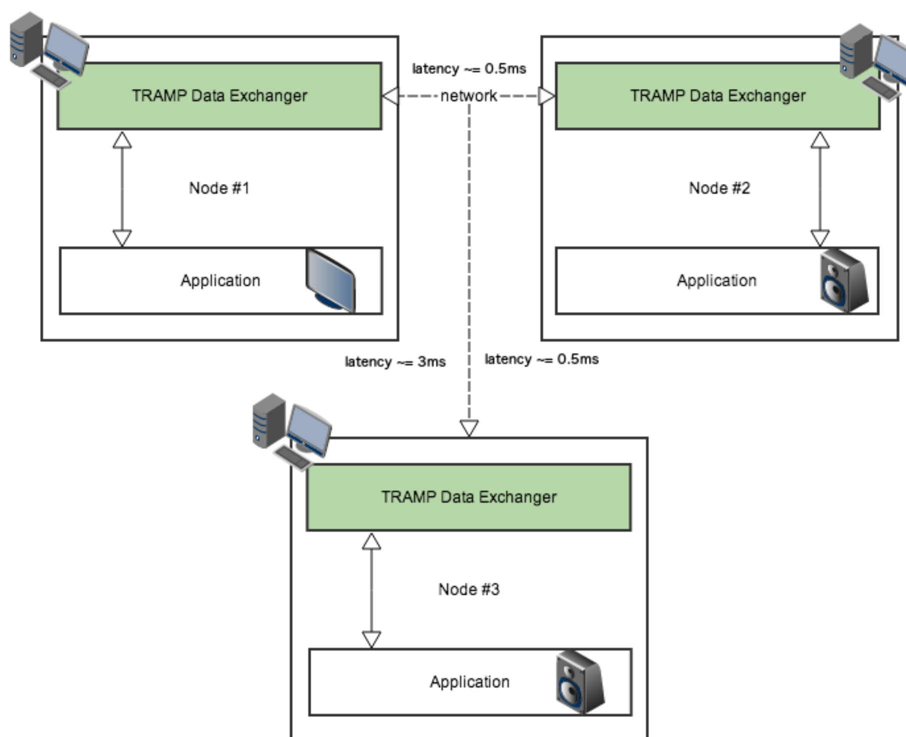


Figure 5.4: Setup for comparing the plugin to the UDP Plugin

Evaluation setup

We illustrate the setup in Figure 5.4. For the sake of simplicity, there are three machines in this setup. That is because it is the minimal number of machines we need to use in order to demonstrate our concept of building distribution trees.

When using the UDP plugin we expect the *Node #1* to broadcast the UDP stream to the nodes *#2* and *#3*. When using TRAMP plugin, we expect the distribution tree to be built based on the latency in the network. In that case the *Node #1* sends the audio stream to the *Node #2* and the *Node #2* also replicates the stream to the *Node #3*.

We have used the *Traffic control* utility available on UNIX platforms, which comes with the *tc* command line tool. With this tool, it is possible to emulate higher

latency, buffer loss in the network for certain interfaces. It is also possible to modify the behaviour of the network interface for a specific IP address. We have used this functionality to emulate increased latency between *Node #1* and *Node #3*.

In this setup we have inspected whether the distribution tree is built based on latency. Figure 5.4 shows that if *Node #3* wants to subscribe to data located on *Node #1* with the lowest latency possible, *Node #2* has to replicate the data.

We have published a video stream by *Node #1* and subscribed to it with Nodes *#2* and *#3* respectively.

5.5 Results

This section presents the results of our evaluation. At first we present the measurement results. Later we compare our plugin to the GStreamer UDP plugin. Last but not least, we demonstrate that the features of GStreamer can be combined with TRAMP when using our plugin making the application programmer's work easier during development of multimedia applications.

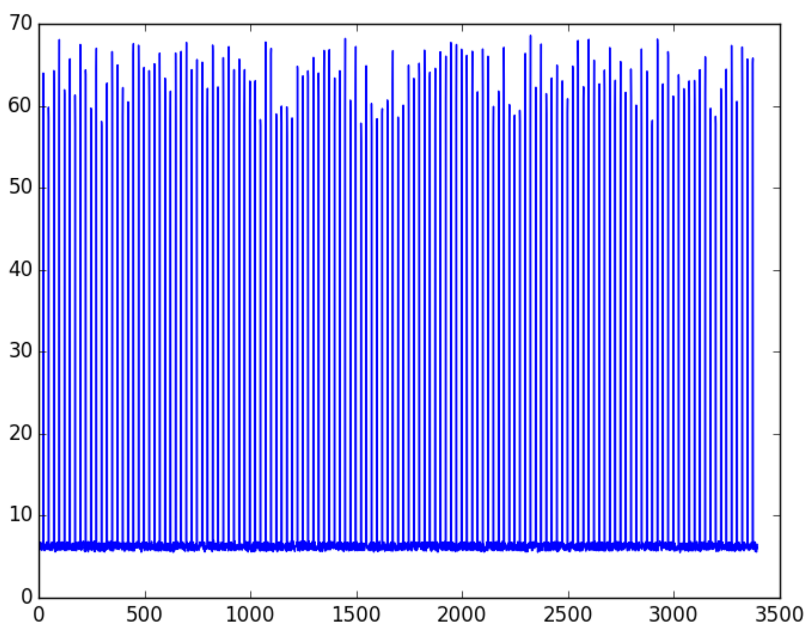


Figure 5.5: Latency

5.5.1 Latency

We evaluate latency using the setup from Figure 5.2. We send a H264 encoded video stream and compare time stamps on two synchronized machines. The results

can be found in Figure 5.5. We illustrate the ID of the sent buffer on the horizontal axis. There is a corresponding latency for each buffer on the vertical axis.

The graph shows that the maximum latency is around 68ms which is less than 100ms we set up as a requirement. Buffers with such latency are exceptional and the higher latency is possibly caused by a context switch.

The network latency between these two nodes is approximately 0.5ms and we measured it using the *ping* tool. The time difference between the data segment entering our plugin and leaving it is within the range of 6-7ms. If we subtract the real network latency we notice that the latency introduced by our plugin is approximately 5.5-6.5ms, which is below the requirement of 100ms.

Most of the time delay we introduce in our plugin is caused by copying the memory block containing multimedia buffer to the shared memory block allocated by TRAMP. The introduced delay is therefore dependent on the maximum size of the buffer. It is important to mention that this 5-6ms delay also includes overhead by the transport layer and a time it takes to do a context switch. This means that our plugin can be used to implement real-time multimedia applications.

5.5.2 Processed buffers per time interval

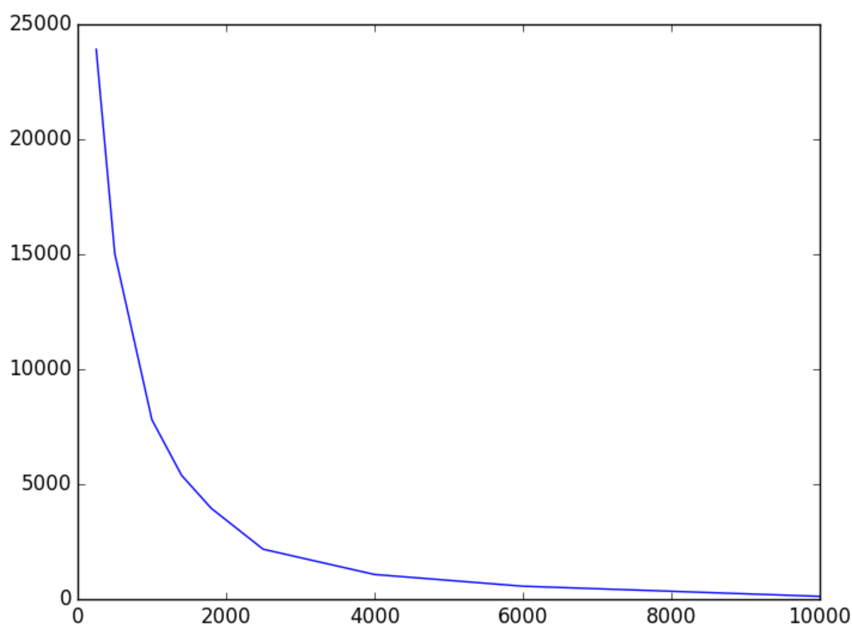
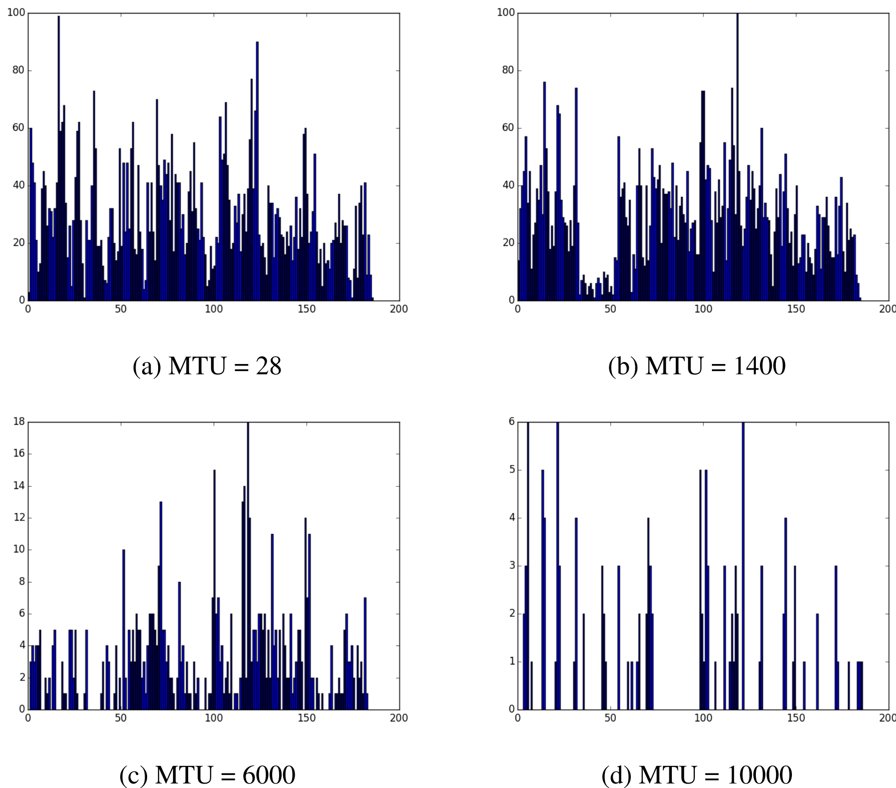


Figure 5.6: Different miss values depend on the value of MTU

The x axis illustrates the MTU value, y axis shows the number of missed buffers per certain time interval.

Desktop computer with high performance

In this setup, we use two pipelines illustrated in Figure 5.2 running on the same Desktop machine described in Table 5.1. We found that our plugin has a problem handling a multimedia stream with a very low MTU. Figure 5.7 shows that lower MTU values mean a higher miss count by the consumer. On the vertical axis we illustrate the number of missed buffers for each second, shown on the horizontal axis. We have performed the test using a video with a duration of 180 seconds.



(a) MTU = 28 (b) MTU = 1400

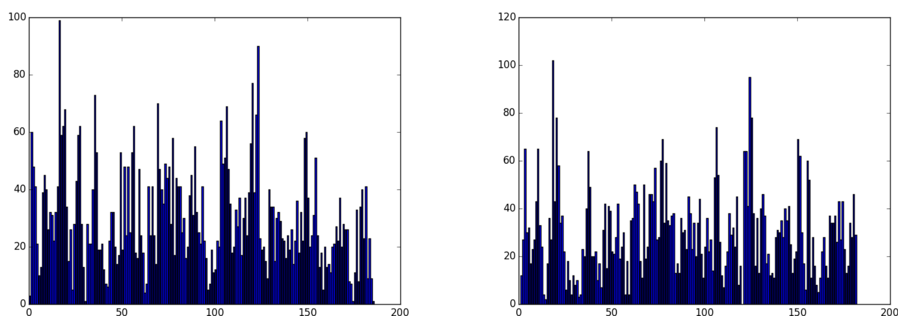
(c) MTU = 6000 (d) MTU = 10000

Figure 5.7: Missed data segments with different values of MTU property

The x axis illustrates the video duration while the y axis shows the number of missed buffers in certain time.

The default MTU value of the *rtph264pay* packetizer is 1400 bytes, while the minimal is 28 bytes. Figure 5.7a and Figure 5.8a show the miss ratio is too high while using these MTU values. This fact prevents us from transmitting packetized H264 encoded video on the same machine, but also from sharing it using TCP connections. As seen in the figures, we would loose too many buffers while using these MTU values, because the receiver would not have enough time to detect the memory change of the sent buffer. We have managed to send good quality video stream starting with the values of MTU 6000 and higher.

Figure 5.6 shows the impact of MTU size on the number of missed buffers by a consumer. We have measured the number of missed buffers in the video sample of 180 seconds. We have constructed this graph based on measurements with the following MTU values: 250, 500, 1000, 1400, 1800, 2500, 4000, 6000 and 10000 bytes.



(a) MTU = 1400, buffer size = 22 500 bytes (b) MTU = 1400, buffer size = 55 000 bytes

Figure 5.8: Number of missed data segments with size of the shared buffer

The x axis illustrates the video duration while the y axis shows the number of missed buffers in certain time.

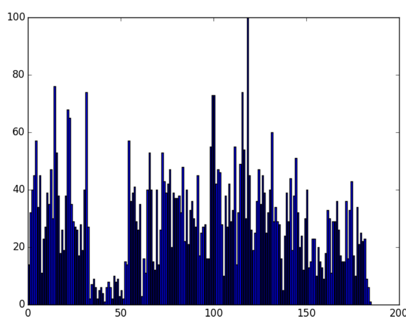
Maximum size of a shared buffer In this part, we inspect whether the maximum size of the shared buffer in TRAMP influences the miss ratio at different MTU values. We have run all of the experiments with a buffer size of 22,500 bytes. In the second run we have doubled the buffer size to 55,000 bytes and run all measurements again. We found out that the miss ratio is approximately within the same range. We provide Figure 5.8 where we compare two runs with MTU set to 1,400 bytes but with a different buffer size. We have chosen to present this specific value because it is the default MTU value of the *rtph264pay* element.

Desktop computer with low performance

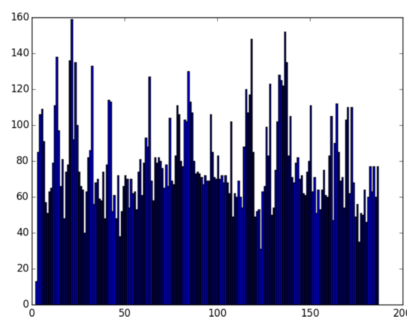
In this part of the evaluation, we try to find out whether performance of the hardware has an impact on the number of reads missed by the consumer. We have re-run all the experiments with the same values on different hardware. Figure 5.9 shows that the number of missed buffers decreases with the speed of the processor.

5.5.3 CPU/Memory load

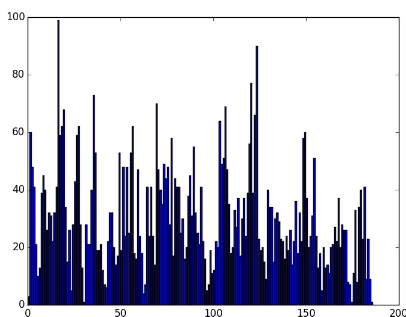
We have separately evaluated the source element and sink element of our plugin with respect to the CPU and memory load.



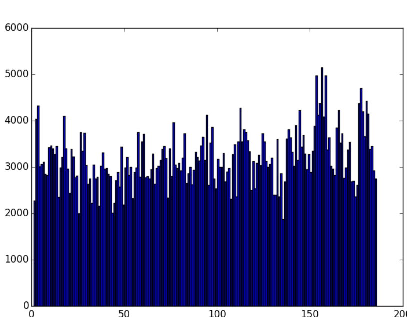
(a) MTU = 1400, High performance node



(b) MTU = 1400, Low performance node



(c) MTU = 28, High performance node



(d) MTU = 28, Low performance node

Figure 5.9: Number of missed data segments - comparing nodes with different performance capabilities

The x axis illustrates the video duration while the y axis shows the number of missed buffers in certain time.

CPU load

We have evaluated the CPU load on the Desktop computer running Fedora described above. Figure 5.10 shows the CPU load of the consumer who is replicating the data for the other node in the network.

We can observe that an element in the role of a producer uses around 10% of the processor's resources. This 10% is mostly consumed by the *memcpy* call responsible for copying GStreamer buffers into shared memory segments. This measurement shows that the producing element does not cause very high CPU load.

One of the cores is constantly using 100% of the resources. That is caused by the implementation of the source element which is constantly checking for memory change inside the infinite loop. TRAMP does not currently have any mechanism to asynchronously notify the consumer about any processes of the incoming data buffer. Therefore naive pooling for memory change inside an infinite loop is the only choice for now. Implementation of a solution which would notify the consumer

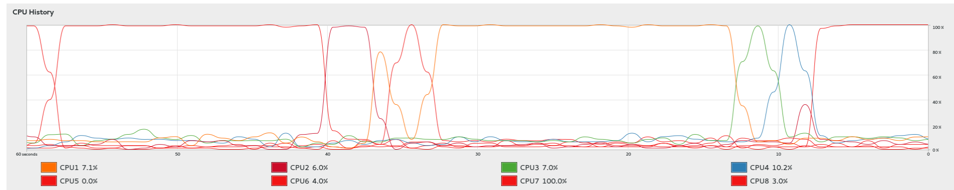


Figure 5.10: CPU load of the consuming/replicating process

about any processes is left for future work.

Memory load

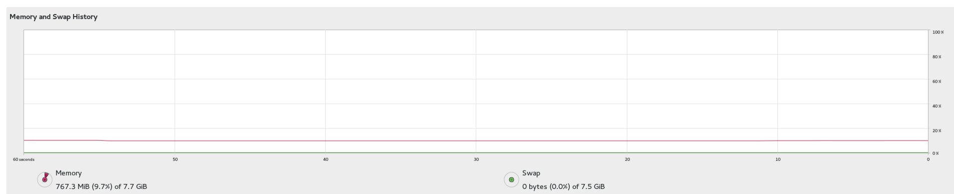


Figure 5.11: Memory load of the consuming/replicating process

We have not found any issues regarding the design or the implementation of our plugin when it comes to memory usage. Figure 5.11 illustrates the memory load of the consuming process which also acts as a replicator. The graph also includes the memory load by TRAMP daemon. We have measured the memory load using the *gnome-system-monitor*. Memory usage was constant during the whole experiment. All checks performed by the *valgrind* did not report any memory leaks.

5.5.4 Comparison with the UDP plugin

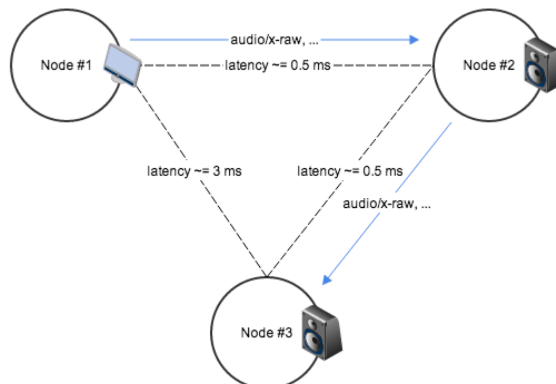


Figure 5.12: Distribution tree built with TRAMP

In this subsection, we present the results of our experiment described in Subsection 5.4.5. Figure 5.13 shows the network traffic of *Node #2* during the experiment, while Figure 5.10 shows the CPU load of *Node #2*. Both figures demonstrate that the distribution tree is built by a TRAMP daemon.

We have measured the corresponding network and CPU loads of *Node #1* and *Node #3* and their bandwidth correspond to the Figure 5.13.

In the case of the UDP Plugin, the stream is broadcasted using UDP directly. That means we have managed to deliver data with lower latency by building a distribution tree. A distribution tree that has been built is shown in Figure 5.12.

This means that our plugin can deliver real-time data within deadlines in situations where UDP protocol would be inefficient and consume too much bandwidth.

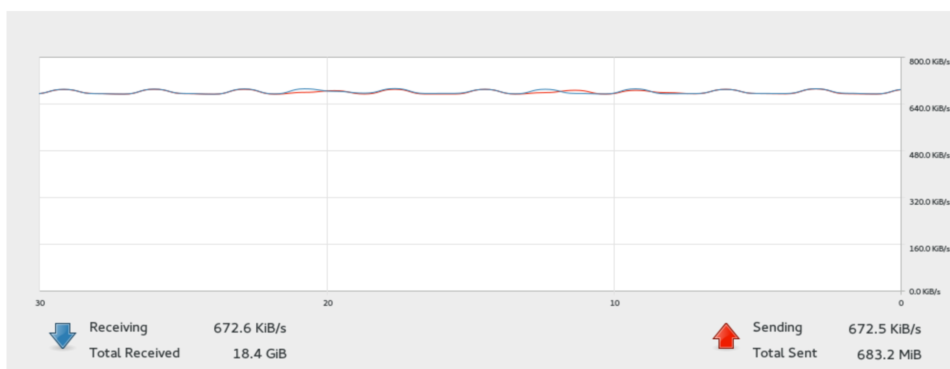


Figure 5.13: Network history of the consumer - replicator

5.5.5 User friendliness

In this Subsection, we present a couple of pipeline setups which can be easily built using our multimedia plugin. By presenting these examples we point out that an application programmer with little knowledge of the GStreamer framework can easily write distributed multimedia applications. Thanks to the GStreamer command line tools, it is possible to construct and run a pipeline using a single shell command. That means an end-point user can interact with the TRAMP Data Exchanger in a number of various ways with no need to compile any code. Changing the behavior of the pipeline is as easy as modifying the property of a chosen element or adding/removing an element to/from the pipeline.

Setup 1

In the first setup we prove that our plugin can play multimedia content on several machines in real-time. This setup includes three nodes. The first node is the node where the video file is located and serves as a producer. The other two nodes are in the role of consumers - the second node playing the video stream, the third node playing the synchronized audio stream.

We illustrate the setup in Figure 3.1 and present the pipelines in Listing 5.1.

Listing 5.1: GStreamer pipelines used in Setup #1

```
// Node #1 providing other nodes with data
$ gst-launch-1.0 filesrc location=/input_movie.avi ! decodebin
    name=dec ! trampsink label=videostream dec. ! audioconvert !
    audioresample ! trampsink label=audiostream

// Node #2 playing video stream
$ gst-launch-1.0 trampsrc label=videostream ! videoconvert !
    autovideosink

// Node #3 playing audio stream
$ gst-launch-1.0 trampsrc label=audiostream ! autoaudiosink
```

During the development, we have used this setup to test the *TRAMP Data Exchanger*'s ability to handle multimedia data. We have also managed to use this setup as a simple broadcasting solution. We managed to connect other machines to this setup resulting in decreased latency because of the distribution tree built by TRAMP.

Running this setup using a command line is just one way to construct these pipelines. In case more functionality is needed, a separate C program can be linked with the shared library of our plugin and GStreamer.

Setup 2

In order to present more interactive example, in the second setup we present a simple video conferencing program built using our system.

In this setup, we connect two personal computers. The first one is the producer recording the video input using a web camera. The second node receives this stream and plays it to the user or saves it to a file. Once the TRAMP issue mentioned in Subsection 3.5.3 is resolved, they can subscribe to each other's multimedia stream and create a fully functional video conference possibly with more nodes interacting.

Listing 5.2 shows how video and audio streams recorded on the first node can be saved to a file in real-time on the other node.

Listing 5.2: GStreamer pipelines used in Setup #2

```
//  Node #1 recording audio/video stream and providing it for the
    consumers
$ gst-launch-1.0 v4l2src ! "video/x-raw,format=(string)I420,width
    =320,height=240,framerate=(fraction)25/1" ! queue ! mux.
    autoaudiosrc ! audioconvert ! "audio/x-raw,rate=44100,channels
    =2" ! queue ! mux. avimux name=mux ! trampsink

// Node #2 receiving the stream and saving it as an .avi file
```

```
$ gst-launch-1.0 trampsrc ! filesink location=file.avi
```

5.6 Discussion

In the previous section, we have presented the results of our experiments. We have fulfilled the goals we set up in the first section of this chapter.

Goal 1 We have tested our multimedia plugin with various types of multimedia content and we have managed to simultaneously play this content on several machines in the real-time. We have managed to share multiple different multimedia streams including real-time audio and video content. We identified issues that need to be solved to send encoded and packetized multimedia content and we have addressed the most important one in Subsection 5.4.2. Resolving this issue is left for future work.

By sharing various types of multimedia streams, we help to evaluate *TRAMP Data Exchanger* with the real multimedia data. As of today, it has been tested with non-multimedia content only, therefore using multimedia data during an evaluation is beneficial for the development of this data sharing platform.

Goal 2 We have evaluated the latency overhead which is within the range of 100ms, which fulfils our requirements for multimedia applications. When sending non encoded multimedia streams, we have managed to handle more than 6Mbit/s bitrate, which is sufficient for transmitting high-quality video streams. We have managed to achieve these results with the hardware presented in Table 5.1.

We have evaluated the CPU and memory load of both sink and source elements while producing, consuming and replicating data. These values were within the acceptable range and we have not experienced any unexpected behavior causing bad performance.

Goal 3 We have listed two examples of the GStreamer pipeline with the usage of our plugin. We have shown that the GStreamer framework allows us to modify the pipeline on the command line. That makes it easier to test and evaluate the TRAMP Data Exchanger. We have also stated several additional advantages of using GStreamer like time-stamping or automatic generation of GUI for a pipeline. Since pipeline design gives us the power to create endless number of pipeline variations, we claim that supporting TRAMP in this framework makes TRAMP easier to use.

Goal 4 We have evaluated our plugin on two operating systems - Fedora release 20 (Heisenbug) and OS X Mavericks. Even though both of these systems are UNIX based, they differ in a high number of properties. Since we have focused on identifying design and performance issues, evaluation of our plugin on Windows operating

system and mobile platforms is left for future work. During the implementation, we have used libraries and tools which are available for these platforms. Therefore adding support for these operating systems is not difficult.

Goal 5 We compared the GStreamer TRAMP plugin to the UDP plugin because they both deal with the same task while using different underlying protocols. The task is to efficiently deliver data from one pipeline to another one. We have used the scenario with three connected nodes. We presented how latency-based distribution trees are built per data segment in order to decrease latency. We have verified the functionality of TRAMP data replication by inspecting CPU and network load while playing an audio stream.

Chapter 6

Conclusions

In this chapter, we conclude our work and discuss the results we achieved. In Section 6.1 we present our contributions. In Section 6.2 we discuss possible improvements that are left for future work.

6.1 Contribution

We have designed and implemented a tool for handling multimedia content in a distributed environment. The main goal of our work was to integrate data sharing platform with multimedia framework in order to provide application programmers with efficient development tool for implementing distributed multimedia applications. We have analysed several data sharing platforms and multimedia frameworks and picked those that fulfilled our requirements presented in Section 1.3. We have chosen the *TRAMP Data Exchanger* as an underlying framework for providing data in a distributing environment and GStreamer as a multimedia framework which wraps the functionality of this data sharing platform.

The most important result of our work is a GStreamer plugin for the *TRAMP* data sharing platform. We aimed to make it easier for an application programmer to create distributed multimedia applications using TRAMP and allow them to take advantage of valuable features of GStreamer at the same time. During the whole development, we kept in mind that wrapping the functionality of TRAMP into the multimedia framework should not decrease its performance or other features like portability.

We have implemented two versions of this plugin. The first one is the simple implementation of our design making it possible to share real-time multimedia streams between several machines. The second implementation aims to support advanced features of GStreamer. Currently, we have not managed to fix all of the *TRAMP Data Exchanger* issues. This restricted us to successfully test and evaluate the first implementation only.

By designing and implementing this solution, we have helped to reveal relevant issues and bugs in the *TRAMP Data Exchanger* program which is still being

developed. In Section 3.5, we have described and analysed these issues and offered possible solutions. We have fixed some of those issues in the *TRAMP Data Exchanger* implementation.

The *TRAMP Data Exchanger* is currently being developed and has not yet been tested as part of a multimedia application. We have made it possible to easily test the *TRAMP Data Exchanger* with different types of multimedia streams and managed to send real-time video and audio streams using this platform while keeping latency overhead under 100ms. We have managed to share multimedia streams using our plugin at different bit rates up to 6Mbit/s. We have demonstrated that when using our plugin, an application programmer can take advantage of latency-based distribution trees that are built by the TRAMP Data Exchanger. That allows application programmers to share multimedia streams in scenarios where low latency is required and using latency-based distribution trees is more efficient than broadcasting using the UDP protocol.

6.2 Future work

There are two categories of problems that are left for future work. The first category includes the problems and fixes in the *TRAMP Data Exchanger*. The second category consists of problems with multimedia plugin itself.

TRAMP Data Exchanger The only fatal problem is addressed in Subsection 5.4.2 that is the number of processed buffers per time interval. Since this problem has not yet been fixed, it did not allow us to use our plugin with compressed and packetized streams and only allowed us to test our platform with uncompressed multimedia data causing increased bandwidth as described in the Evaluation chapter.

Other issues that need to be fixed are mostly minor problems that would allow better performance or allow our system to be used in even more use cases. Rebalancing a distribution tree in case of latency change would significantly increase the performance and overall user experience. It would also solve the problem addressed in Subsection 3.5.3 making it possible to create a fully functional video conference where two users subscribe to each others video streams as described in our evaluation.

TRAMP GStreamer plugin Once these *TRAMP Data Exchanger* issues are fixed, implementation of the plugin supporting more GStreamer features will be possible. What is left for future work is completing the details of our second implementation including the support for dynamic negotiation and various types of GStreamer events and queries.

Bibliography

[1] Documentation: Vlc modules loading. https://wiki.videolan.org/Documentation:VLC_Modules_Loading/, October 2012. [Online; accessed 22-February-2014].

[2] Hacker Guide/How To Write a Module. https://wiki.videolan.org/Hacker_Guide/How_To_Write_a_Module/, 2013. [Online; accessed 23-February-2014].

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. *SIGPLAN Not.*, 25(3):168–176, February 1990.

[4] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. Technical report, Pittsburgh, PA, USA, 1993.

[5] Robert D. Bjornson. *Linda on distributed memory multiprocessors*. PhD thesis, New Haven, CT, USA, 1993. UMI Order No. GAX93-29333.

[6] J. Dean Brock, Rebecca F. Bruce, and Marietta E. Cameron. Changing the world with a raspberry pi. *J. Comput. Sci. Coll.*, 29(2):151–153, December 2013.

[7] Nicholas Carriero and David Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems (TOCS)*, 4(2):110–129, 1986.

[8] Python Software Foundation. Python programming language. https://www.python.org/. [Online; accessed 29-November-2013].

[9] Python Software Foundation. Tuples and sequences. https://docs.python.org/2/tutorial/datastructures.html#tuples-and-sequences. [Online; accessed 29-November-2013].

[10] Eric T Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns and Practices*. Addison-Wesley Professional, 1999.

[11] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[12] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[13] Hans Vatne Hansen, Francisco Velázquez-García, Vera Goebel, and Thomas Plagemann. Efficient data sharing for multi-device multimedia applications. In *Proceedings of the Workshop on Multi-device App Middleware*, Multi-Device '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[14] Raj Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 2008.

[15] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 13–21, New York, NY, USA, 1992. ACM.

[16] V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter. The architecture of a linda coprocessor. *SIGARCH Comput. Archit. News*, 16(2):240–249, May 1988.

[17] Kai Li. *Shared virtual memory on loosely coupled multiprocessors*. PhD thesis, New Haven, CT, USA, 1986. AAI8728365.

[18] The Linux man-pages project. Top(1) man page. `http://man7.org/linux/man-pages/man1/top.1.html`, September 2002.

[19] The Linux man-pages project. Kill(2) man page. `http://man7.org/linux/man-pages/man2/kill.2.html`, September 2013.

[20] The Linux man-pages project. Malloc(3) man page. `http://man7.org/linux/man-pages/man3/malloc.3.html`, December 2013.

[21] The Linux man-pages project. Memcmp(3) man page. `http://man7.org/linux/man-pages/man3/memcmp.3.html`, March 2013.

[22] The Linux man-pages project. Memcpy(3) man page. `http://man7.org/linux/man-pages/man3/memcpy.3.html`, March 2013.

[23] The Linux man-pages project. Read(2) man page. `http://man7.org/linux/man-pages/man2/read.2.html`, February 2013.

[24] The Linux man-pages project. Write(2) man page. `http://man7.org/linux/man-pages/man2/write.2.html`, January 2013.

[25] The Linux man-pages project. Valgrind(1) man page. `http://man7.org/linux/man-pages/man1/valgrind.1.html`, April 2014.

[26] David L. Mills. On the accuracy and stablility of clocks synchronized by the network time protocol in the internet system. *SIGCOMM Comput. Commun. Rev.*, 20(1):65–75, December 1989.

[27] The GNOME Project. Gobject — the base object type. https://developer.gnome.org/gobject/unstable/gobject-The-Base-Object-Type.html, 2005.

[28] Steve Baker Richard John Boulton, Erik Walthinsen et al. GStreamer Plugin Writer's Guide (1.2.3). http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/pwg.pdf. [Online; accessed 23-December-2013].

[29] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*, volume 2. Prentice Hall, 2002.

[30] Wim Taymans, Steve Baker, Andy Wingo, et al. GStreamer Application Development Manual (1.2.3). http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/manual.pdf. [Online; accessed 23-December-2013].

[31] GStreamer Team. Gstreamer: open source multimedia framework. http://gstreamer.freedesktop.org/. [Online; accessed 04-November-2013].

Glossary

API Application Programming Interface.

CPU Central Processing Unit.

DBUS Desktop Bus.

DSM Distributed Shared memory.

HTTP Hypertext Transfer Protocol.

MTU Maximum transmission unit.

RPC Remote procedure call.

RPM Red Hat Package Manager.

RTP Real-time Transport Protocol.

SHM Shared memory.

SMTPE Society of Motion Picture & Television Engineers.

TCP Transmission Control Protocol.

TRAMP TRAMP Real-time Application Mobility Platform.

UDP User Datagram Protocol.

V4L2 Video4Linux.

Appendix A

Deployment

A.1 Compiling RPM packages

In order to easily use and test our GStreamer plugin, it should be possible to easily install it in the operating system.

We have created two RPM packages - the *tramp-data-exchanger*, and the *gstreamer-plugins-tramp* package, so the project can be easily installed and used on any RPM-based Linux distribution. These include for example Fedora, Red Hat Linux, Mandriva and SUSE Linux.

By creating RPM packages, we wanted to prove that the system can be packaged and used on a personal computer. Creating other formats such as *.deb* and *.exe* is possible, but it is out of the scope of this paper.

RPM package tramp-data-exchanger contains TRAMP Data Exchanger and installs two important command line utilities.

- trampd - TRAMP Data Exchanger Daemon used to start a DSM daemon process
- shm-clear.sh - Bash script used to clear unused DSM data segments on the machine

RPM package gstreamer-plugins-tramp is a package which installs shared libraries so we can use our elements in the GStreamer console pipeline as shown in Listing A.1. It is packaged using RPM and *autotools*. It also contains header files needed to compile C program with the usage of our plugin.

Listing A.1: Running GStreamer TRAMP plugin

```
machine 1 $ gst-launch filesrc ! ogg-demuxer name=t ! vorbis-
    decoder ! trampsink label=tramp_label t. ! theora-decoder !
    video-sink
machine 2 $ gst-launch trampsrc label=tramp_label ! audio_sink
```

A.2 Installation

We can install RPM files on any RPM-based Linux distribution. For instance on Fedora, we can install the system by adding a repository and installing *gstreamer-plugins-tramp*. Yum takes care of dependencies such as *tramp-data-exchanger* and installs the plugin as shown in Listing A.2.

Listing A.2: Installing TRAMP GStreamer plugin on Fedora

```
# rpm --import http://janvor.at.ifi.uio.no/JAN-VORCAK-GPG-KEY
# yum-config-manager --add-repo http://janvor.at.ifi.uio.no/tramp-
    repo
# yum install gstreamer-plugins-tramp
```

A.3 Usage

In order to use TRAMP GStreamer plugin, you just need to connect *trampsink* and *trampsrc* elements to the pipeline as shown in Listing A.1.

If these two processes are run on different machines, we need to make sure that TRAMP Data Exchanger is running. It can be executed with *trampd* command.

Appendix B

Source code

The source code of the improved version of *TRAMP Data Exchanger* can be downloaded as shown in Listing B.1.

Listing B.1: Downloading our version of TRAMP Data Exchanger

```
$ wget http://janvor.at.ifi.uio.no/tramp-daemon.tar
$ tar -xvpf tramp-daemon.tar
$ cd tramp_data_exchanger
```

In order to fetch the source code of our GStreamer plugin, use the commands from Listing B.2.

Listing B.2: Fetching the source code of GStreamer TRAMP plugin

```
$ http://janvor.at.ifi.uio.no/gst-plugin.tar
$ tar -xvpf gst-plugin.tar
$ cd gst-plugin/src
```