Communication Analysis of Programs by Assembly Level Execution Traces

by

Huihui(Nora) Huang
B.Sc., Nanjing University of Aeronautics and Astronautic, 2003
M.Sc., Nanjing University of Aeronautics and Astronautic, 2006

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Huihui(Nora) Huang, 2018
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Communication Analysis of Programs by Assembly Level Execution Traces

by

Huihui(Nora) Huang
B.Sc., Nanjing University of Aeronautics and Astronautic, 2003
M.Sc., Nanjing University of Aeronautics and Astronautic, 2006

Supervisory Committee

Dr. Daniel German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

Supervisory Committee

Dr. Daniel German, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Departmental Member
(Department of Computer Science)

Dr. Outside Member, Outside Member
(Department of Not Same As Candidate)

ABSTRACT

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

The internet grows incredibly fast in the past few year. More and more computers are connected to it in order to get service or provide service. The internet as a powerful platform for people to share resource, meanwhile, introduces the risk to computers in the way that it enable the exploit of the vulnerability of the software running on it.

Accordingly, the emphasize place on computer security particularly in the field of software vulnerabilities increase dramatically. It's important for software developers to build secure applications. Assurance about the integrity and security of the software are expected from the vendors of the software. Unfortunately, building secure software is expensive. The vendors usually comply with their own quality assurance measures which focus on marketable concerns while left security in lower priority or even worse totally ignore it. Therefore, fully relying on the vendor of the software to secure you system and data is unpractical and risky.

Software security review conducted by a third party is usually more convincing and comprehensive. One approach of software security review is software auditing. It is a process of analyzing the code of the software in form of source code or binary. This auditing can uncover some hard to reveal vulnerabilities which might be misused by the hackers. Identification of these security holes can safe the users of the software from putting their sensitive data and business resources at risk.

Most of the software vulnerabilities are caused by malicious data intrusion. So it is valuable to understand how this malicious data trigger the unexpected behaviors of the system. In most of the case this malicious data is injected by an attacker into the system to trigger the exploit. Nonetheless, this malicious data might go through devious path to ultimately triggers an exploitable condition of the system. In some complicated system, several components which are collaborated programs work together to provide service or functionality. In these situation the malicious data might have passed through multiple components of the system and be modified before it reach the vulnerable point of the system. As a consequence, the flow of data throughout the system's different programs is considered to be one of the most important attribute to analysis during the security review.[7]

The data flow among various programs with in the system or across different systems helps to understand how the system work as well as disclose the vulnerabilities of the system as stated before. There are multiple mechanism to grab the data across programs. And the methods for the grasp of this data flow is essential and can affect the analysis result greatly. For instance, packet capture by some sort of sniffers in the network is considered to be insufficient for security problems detection by the experience security engineer from DRDC(research partner of our group). Instead, dynamic analysis of the programs by capturing and analyzing their execution traces with memory accesses is an relatively accurate to analysis the data transmitted throughout the programs of the system.

In this research, I developed a method to analysis communications between the programs by the analysis of the execution traces of them. I didn't aim at covering all the communication types but only focus on the data exchanging ones. This method should be able to guide the security engineers to investigate the communications of the programs in the circumstance that they have the captured execution traces and want to understand the interaction behavior of the programs. The research is not specified for vulnerabilities detection but generalized for the comprehension of the behavior of the programs.

1.1 Motivation

This project started with an informal requirement from DRDC for visualizing multiple assembly traces to assist their software and security analysis. The literature review and the conversation with DRDC help to clarify the goal and target of this research. In this section, I discuss the need of performing assembly trace investigation for communication analysis. First I explain why security engineers perform assembly trace analysis. Then I elaborate why they need to perform communication analysis at assembly trace level. Out of the answers of these two questions, I conduct this research.

1.1.1 Why Assembly Trace Analysis

Dynamic analysis of program is adopted mainly in software maintenance and security auditing[26], [4], [20]. Sanjay Bhansali et al. claimed that program execution traces with the most intimate detail of a program's dynamic behavior can facilitate the program optimization and failure diagnosis. Jonas Trümper et al. give a example of how tracing can facilitate software-maintenance tasks [22].

The dynamic analysis can be done by using debuggers, however, debuggers would halt the execution of the system and result in a distortion of the timing behavior of the running system [22]. Instead, tracing a running program with instrumentation would provide more accurate run time behavior information about the system.

The instrumentation of the tracing can be in various level, such as programming language or machine language levels. The choice in some how depends on the accessibility to the application. The application access is divided into five categories with variations: source only, binary only, both source and binary access, checked build, strict black box. Binary only category is common when performing vulnerability research on closed-source commercial software.[7] In this case, assembly level provides the possibility of the security review of the software. Unavailability of source is the

On the other hand, since the binary code is actually what is running on the system, it is more representative of what is actually running than the source code. Some bugs might appear because of a compilation problem or because the compiler optimized away some code that was necessary to make the system secure. The piece of code list below is an example in which the code line of code resetting the password before the program end would be optimized away by GNU Compiler Collection(GCC) due to it is not used later. This made the user's password stayed in memory, which is considered as a security flaw. However, by looking at the source code didn't reveal that problem.

Listing 1.1: Password Fetching Example

```
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
int main(){
  string password ="";
  char ch;
  cout << "Enter␣password";
  ch = _getch();
  while(ch != 13){//character 13 is enter
    password.push_back(ch);
    cout << '*';
    ch = _getch();
  }
  if(checkPass(password)){
   allowLogin();
  }
  password ="";
}
```

1.1.2 Why Communication Analysis with Assembly Traces

Programs nowaday do not alway work isolately, many software appear as reticula collaborating systems connecting different modules in the network[17] which make the discovery of vulnerabilities even harder. The communication and interaction between modules affect the behavior of the software. Without regarding to the synergy information , analysis of the isolated execution trace on a single computer is usually futile. The tracing data flow process is essential to reviews of both the design and implementation of software.

Many network sniffer such as Wireshark[6] and Tcpdump[21] can help to capture the data flow across the network which make the systematic analysis possible. However, it is claimed as the insufficient method due to the fact that security problems can occur even if the information sent is correct. Therefore, analyzing the communications with transmitted data in instruction and memory access level is a solid way to evaluation the system.

Shameng Wen et al. argued that fuzz testing and symbolic execution are widely applied to detect vulnerabilities in network protocol implementations. They present their model-guided approach to detect vulnerabilities in the network protocol implementation. Their work emphasized model which guide the symbolic execution for the fuzz testing while ignoring the analysis of the output, which can be the execution traces, from the execution. [24] Further more, their work focus only on the network protocol implementation but not generalized to all communication method, including network protocol and inter process communications.

Besides vulnerabilities detection and security reason, communication analysis with assembly traces can also be a way to learn how the work is performed by the system or sometime validate a specification of it. Our research partner DRDC provided use cases of communication analysis which related to their work with embedded system. These systems often have more than one processor, each specialized for a specific task, that coordinate to complete the overall job of that device. In other cases, the embedded device will work with a normal computer and exchange information with it through some means (USB, wireless, etc.). For instance, the data might be coming in from an external sensor in an analog form, transformed by a Digital Signal Processor (DSP) in a device, sent to a more generic processor inside that device to integrate with other data then send wirelessly to an external computer. Being able to visualize more than one trace would help them follow the flow of data through the system.

1.2 Approach

The approach I elaborate in this section is not a forthright process. Instead, it is a back and forth one, for example the implementation changed several times with the changes of the model, and the models was modified based on the understanding throughout the implementation. Here I simply my research approach by listing the key factors in each step while ignoring route of it.

This research requires background knowledge in software security, program communication mechanism and implementation, assembly execution traces. I acquire the software security knowledge basically from literature reviews. It helps me to grab the essential concept of software vulnerabilities and their categories, understand some facilities for vulnerabilities detection and software maintenance in the perspective of security. After that, I was convinced that communication analysis in assembly trace level would benefit software security engineers to understand to behavior of the software and detect software vulnerabilities.

In order to analysis the communication of programs, I had to know how the communication works. For this purpose, I started the investigation from a piratical experiment by writing example simple programs with the Windows API and run them locally in my desktop. By understanding their behavior and the reading of the Windows API documentation, I abstracted the communication model which is not operating system specific.

The assembly trace model was build on the generalization of the trace format provided by our research partner, DRDC. I don't have the access to their home-made assembly tracer which is based on PIN[12]. Fortunately, they provides a comprehensive document about the format of the captured trace and example traces to me. With these, I grasped the constructive view of the assembly execution trace. Further more, in the present of some dynamic software analysis works [10], [15], [19], [1], [2] and [22], it is certain that, some other tools can also capture the required information in assembly level for communication analysis. This supports the generalization of the trace format and the abstraction of the dual_trace modeling.

The implementation of the prototype and the communication identification algorithms are develop in parallel. The high level identification algorithm and the specific algorithms for named pipe communication methods were abstracted based on the implementation, while the others are developed theoretically.

Among the two experiments, the first simpler one was provided directly by DRDC with their initial requirement, while the second one was designed by me. In both experiments, DRDC conducted the program execution and trace capture on their environment while I performed the analysis locally with Atlantis on my desktop with the captured traces and corresponding .dll.

1.3 Thesis Organization

In Chapter 2, I summarize the related background information and knowledge needed to understand or related to this work including security and vulnerability, program communication mechanisms, program execution trace tools, and Atlantis.

Chapter 3 depicts the model of the communication between two programs and the model of the dual_trace which contains two execution traces of two interacting program. The communication model defines the communication in the context of trace analysis as well as discusses the properties of the communications. The dual_trace model not only represents the original format of the execution traces, but also abstract the elements from the original format and matches them to the elements in the communication model.

Chapter 4 describes the algorithms I developed for the communication analysis of the dual_trace including the high level communication identification algorithm and the detail ones such as event and stream filtering algorithms, stream matching algorithms and data verification algorithms.

To provide more concrete idea, I present, in chapter 5, the implemented communication identification feature prototype. This prototype was built on top of Atlantis[11], an assembly execution trace analysis environment.

In chapter 6, I present two detailed experiments with dual_traces of programs using named pipe for communication on the implemented prototype. Notably, the result shows the communications are correctly identified.

Finally, In chapter 7, I conclude the result of this research and outline the possible future work.

Chapter 2

Background

In this section, I summarize the background knowledge or information that related to this work. First I generally describe software vulnerability. Second, I discuss the general definition of communication among programs and their categorization. Third, I introduce some tools for assembly level program debugging and analysis. Finally I introduce Atlantis, the existing assembly level execution trace analysis environment, on which the implementation of this work based.

2.1 Software Vulnerability

Software vulnerability detection is one of the use cases of the communication analysis with assembly level execution traces. The understanding of fundamentals of software vulnerability is necessary to comprehend some implicit concepts or design intention through out this thesis.

Vulnerabilities, from the point of view of software security, are specific flaws or oversights in a program that can be exploited by attackers to do something maliciousexpose such as modify sensitive information, disrupt or destroy a system, or take control of a computer system or program. They are considered to be a subset of bugs. Input and Data Flow, interface and exceptional condition handling where vulnerabilities are most likely to surface in software and memory corruption is one of the most common vulnerabilities. The awareness of these two facts would make the security auditing and vulnerabilities detection have more clear focus. [7]

2.2 Program Communications

Programs can communicate with each other via diverse mechanisms. The communication happens among processes is known as inter-process communication. This refers to the mechanisms an operating system provides the process to share data with each other. It includes methods such as signal, socket, message queue, shared memory and so on.[9] This communications can happen over network or inside a device. Based on their reliability, the communication methods can be simply divided into two categories: reliable communication and unreliable communication. In this work, communication methods belong to all both categories has been covered. However, I only discuss the message based communication methods while leave the control based communication, like remote function call for the future.

2.3 Program Execution Tracing in Assembly Level

The communication analysis discuss throughout this thesis is based on the assembly traces. Thurs capturing of the execution traces became a prerequisite of this work. DRDC has its own home-made tracer, the traces from which are used in the experiments of this research. However, the model and algorithms developed in this research is not limited with this specific home-made tracer. Any tracer that can capture sufficient information according to the model can serve this purpose.

There are many tools that can trace a running program in assembly instruction level. IDA pro [8] is a widely used tool in reverse engineering which can capture and analysis system level execution trace. Giving open plugin APIs, IDA pro allows plugin such as Codemap [13] to provide more sufficient features for ”run-trace” visualization. PIN[12] as a tool for instrumentation of programs, provides a rich API which allows users to implement their own tool for instruction trace and memory reference trace. Other tools like Dynamic [3] and OllyDbg[25] also provide the debugging and tracing functionality in assembly level.

2.4 Atlantis

Atlantis is a trace analysis environment developed in Chisel. It can support analysis for multi-gigabyte assembly traces. There are several features distinct it from all other existing tools and make it particularly successful in large scale trace analysis. These features are 1) reconstruction and navigation the memory state of a program at any point in a trace; b) reconstruction and navigation of system functions and processes; and c) a powerful search facility to query and navigate traces. The work of this thesis is not a extension of Atlantis. But it take advantages of Atlantis by reusing it existing functionality to assist the dual_trace analysis.

Chapter 3

Modeling

In this chapter, I modeled the communication of two running programs. The dual-trace captured from two interacting programs are also modeled in the perspective of communication analysis. The modeling are based on the investigation of some common used communication methods. The communication methods are divided into two categories based on their data transmission properties. This modeling are the foundation to decide how communications being identified from the dual-trace and how to present them to the user. The terminology of using in this chapter can be found in A.

3.1 Communication Categorization and Communication Methods

The goal of this work is to identify the communications from the dual-trace. We need to understand the properties of the communications to identify them. In general, there are two types of communication: reliable and unreliable in the terms of their reliability of data transmission. The reason to divide the communication methods into these two categories is that the data transmission properties of the communications fall in different categories affect the mechanism of the data verification in the identification algorithm. In the following two subsections, I summarize the characteristics of these two communication categories. The communication methods list in Table3.1 will be discussed further to provide more concrete comprehension.

Table 3.1: Communication Methods Discussed in This Work

Reliable Communication	Unreliable Communication
Named Pipes	Message Queue
TCP	UDP

3.1.1 Reliable Communication

A reliable communication guarantees the data being sent by one endpoint of the channel is always received losslessly and in the same order in the other endpoint. For some communication methods, a channel can be closed before all sent data being received. With this property, the concatenated data in the receive stream of one endpoint should be the prefix of the concatenated data in the send stream of the other endpoint(potentially equal). Therefore, comparing the concatenated received data of one endpoint to the concatenated sent data of the other can verify the send and receive data.

3.1.2 Unreliable Communication

An unreliable communication does not guarantee the data being sent always arrive the receiver. Moreover, the data packets can arrive to the receiver in any order. However, the bright side of unreliable communication is that the packets being sent are always arrived as the origin packet, no data re-segmentation would happen. Accordingly, the send and receive data verification can be done by matching the data packet in a receive event to the data packet in a send event on the other side.

3.1.3 Communication Methods

In this section, I describe the mechanism and the basic data transfer characteristics of each communication method in Table3.1 briefly. Moreover, data transfer scenarios are represented correspondingly in diagrams for each communication method.

Named Pipe

A named pipe provides FIFO communication mechanism for inter-process communication. It can be a one-way or a duplex pipe. [14]

The basic data transfer characteristics of Named Pipe are:

- Bytes are received in order

- Bytes sent as a segment can be received in multiple segments(the opposite is not true)
- No data duplication
- If a sent segment is loss, all the following segments will lost(this happen when the receiver disconnect from the channel)

Based on these characteristics, the data transfer scenarios of Named pipe can be exemplified in Figure3.1.

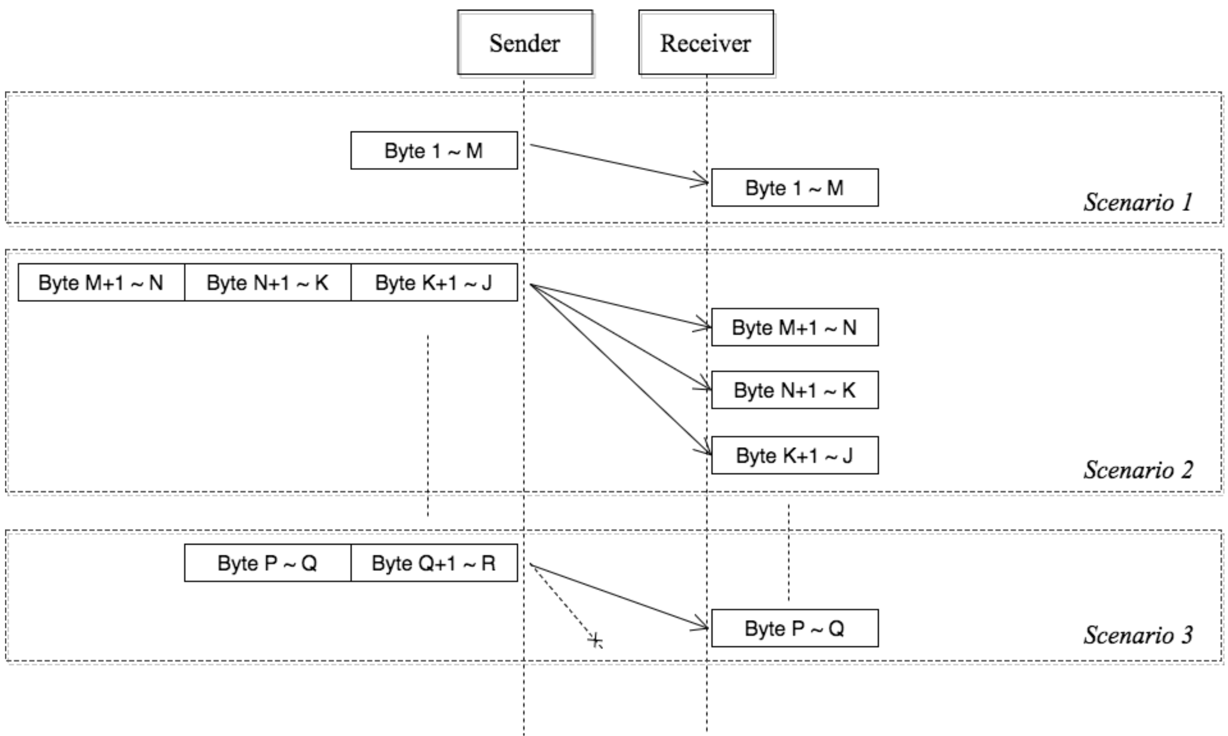

Figure 3.1: Data Transfer Scenarios for Named Pipe

Message Queue

Message Queuing (MSMQ) is a communication method to allow applications which are running at different times across heterogeneous networks and systems that may be temporarily offline can still communicate with each other. Messages are sent to and read from queues by applications. Multiple sending applications can send messages to and multiple receiving applications can read messages from one queue.[18] In this work, only one sending application versus one receiving application case is considered. Multiple senders to multiple receivers scenario can be divided into multiple sender and receiver situation. Both applications of a communication can send to and receive from the channel.

The basic data transfer characteristics of Message Queue are:

- Bytes sent in packet and received in packet, no bytes re-segmented
- Packets can lost
- Packets received in order
- No data duplication

Based on these characteristics, the data transfer scenarios of Message Queue can be exemplified in Figure3.2.

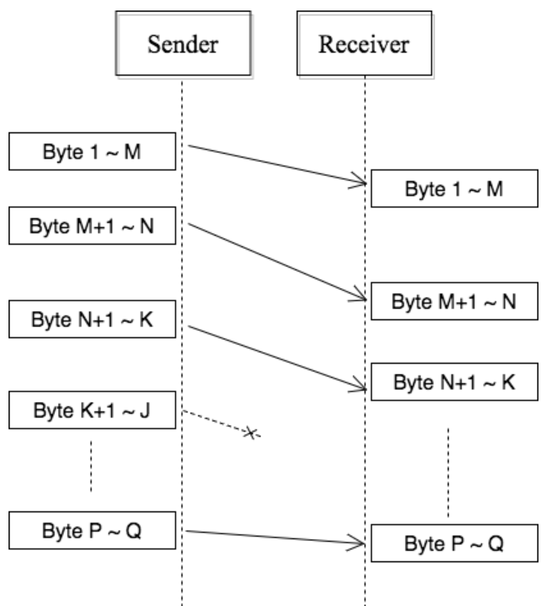

Figure 3.2: Data Transfer Scenarios for Message Queue

TCP

TCP is the most fundamental reliable transport method in computer networking. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts in an IP network. The TCP header contains the sequence number of the sending octets and the acknowledge sequence this endpoint is expecting from the other endpoint(if ACK is set). The re-transmission mechanism is based on the ACK.

The basic data transfer characteristics of TCP are:

- Bytes received in order
- No data lost(lost data will be re-transmitted)
- No data duplication
- Sender window size is different from receiver's window size, so packets can be re-segmented

Based on these characteristics, the data transfer scenarios of TCP can be exemplified in Figure3.3.

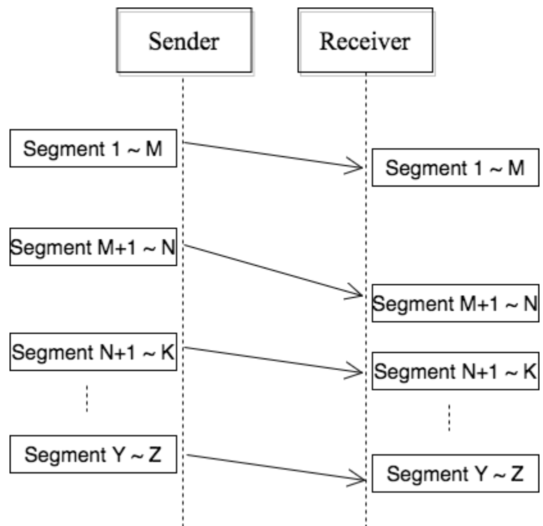

Figure 3.3: Data Transfer Scenarios for TCP

UDP

UDP is a widely used unreliable transmission method in computer networking. It is a simple protocol mechanism, which has no guarantee of delivery, ordering, or duplicate protection. This transmission method is suitable for many real time systems.

The basic data transfer characteristics of UDP are:

- Bytes sent in packet and received in packet, no re-segmentation
- Packets can lost
- Packets can be duplicated
- Packets can arrive receiver out of order

Based on these characteristics, the data transfer scenarios of UDP can be exemplified in Figure3.4.

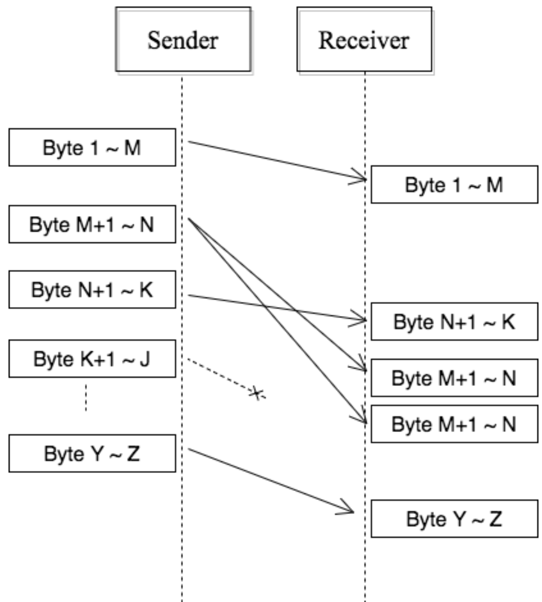

Figure 3.4: Data Transfer Scenarios for UDP

3.2 Communication Model

The communication of two programs is defined in this section. The communication in this work is data transfer activities between two running programs through a specific channel. Some collaborative activities between the programs such as remote procedure call is out of the scope of this research. Communication among multiple programs (more than two) is not discussed in this work. The channel can be reopened again to start new communications after being closed. However, the reopened channel will be treated as a new communication. The way that I define the communication is leading to the communication identification in the dual-trace. So the definition is not about how the communication works but what it looks like. There are many communication methods in the real world and they are compatible to this communication definition.

3.2.1 Communication Definition

In the context of a dual_trace, a communication is a sequence of data transmitted between two endpoints through a communication channel. The endpoints connect to each other using the identifier of this channel. We therefore defined a communication c as a triplet:

$c =< ch, e_0, e_1 >$

where $e0$ and $e1$ are endpoints while ch is the communication channel used (e.g. a named piped located at /tmp/pipe).

From the point of view of traces, the endpoints e_0 and e_1 are defined in terms of three properties: the handle created within a process for the endpoint for subsequent operations(e.g. data send and receive), the data stream received and the data stream sent. Therefore, I define an endpoint e as a triplet:

$e =< handle, d_r, d_s >$

where d_r and d_s are data streams. A data stream is a sequence of events, each sends or receives a package. Each package contains data that is being sent or received (its payload). Hence, we can define a data stream d as a sequence of n packages:

$d = (pk_1, pk_2, ..., pk_n)$

Note that this is the sequence of packages as seen from the endpoint and might be different than the sequence of packages seen in the other endpoint, specially where there is package reordering, loss or duplication.

Each package has several attributes:

- *Relative time(it was sent or received):* In a trace, we do not have absolute time for an event. However, we know when when an event (i.e. sending or receiving a package) has happened with respect to another event. we will use the notation

  $time(pkg)$

  to denote this relative time. Hence, if $i < j$, then

  $time(pk_i) < time(pk_j)$

- *Payload:* Each package has a payload (the data being sent). This payload can be modeled as a string contained in the package.we will use the notation

  $pl(pkg)$

  to denote this payload.

3.2.2 Communication Properties

The properties of the communications can be described based on the definition of the communication.

Properties of reliable communication:

A reliable communication guarantees that the data sent and received between a package happens without loss and in the same order.

For a given data stream, I will define the data in this stream as the concatenation of all the payloads of all the packages in this stream, in the same order, and denote is as $data(d)$.

Given $d =< pk_1, pk_2, ..., pk_n >$, $data(d) = pl(pk_1) \cdot pl(pk_2) \cdot \ldots \cdot pl(pk_n)$

- *Content Preservation:* for a communication:

  $c =< ch, < h_0, dr_0, ds_0 >, < h_1, dr_1, ds_1 >>$

  the received data should always be a prefix (potentially equal) of the data sent:

  $data(dr_0)$ is a prefix of $data(ds_1)$ and

  $data(dr_1)$ is a prefix of $data(ds_0)$

- *Timing Preservation:* at any given point in time, the data received by an endpoint should be a prefix of the data that has been sent from the other:

  for a sent data stream of size m, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size n, $dr =< pkr_1, pkr_2, ...pkr_n >$

  for any $k \in 1..n$, there must exist $j \in 1..m$ such that: pks_j was sent before pkr_k was received:

  $time(pks_j) < time(pkr_k)$

  and

  $data(< pkr_1, pkr_2, ..., pkr_k >)$ is a prefix of $data(< pks_1, pks_2, ..., pks_j >)$

  In other words, at any given time, the recipient can only receive at most the data that has been sent.

Properties of unreliable communication:

In unreliable communication sender and receiver are not concerned with the concatenation of packages. Instead, they treat each package independent of each other.

- *Content Preservation:* a package that is received should have been sent:

  for a sent data stream of size m, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size n, $dr =< pkr_1, pkr_2, ...pkr_n >$

for any $pkr_j \in dr$ there must exist $pks_i \in ds$

We will say that the pkr_j is the matched package of pks_i, and vice-versa, pks_i is the matched package of pkr_j, hence

$match(pkr_j) = pks_i$ and

$match(pks_i) = pkr_j$

- *Timing Preservation:* at any given point in time, packages can only be received if they have been sent

  for a sent data stream of size m, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size n, $dr =< pkr_1, pkr_2, ...pkr_n >$

  for any $k \in 1..n$, $time(match(pkr_j)) < time(pkr_j)$

  In other words, the match of the received package must has been sent before it is received.

In the following two examples, h_0 and h_1 are the handles of the two endpoints e_0 and e_1 of the communications. ds_0, dr_0 and ds_1, dr_1 data streams of the endpoints e_0 and e_1. The string payloads are the strings represented in blue and red in the figures.

Figure3.5 is an example of the reliable communication.

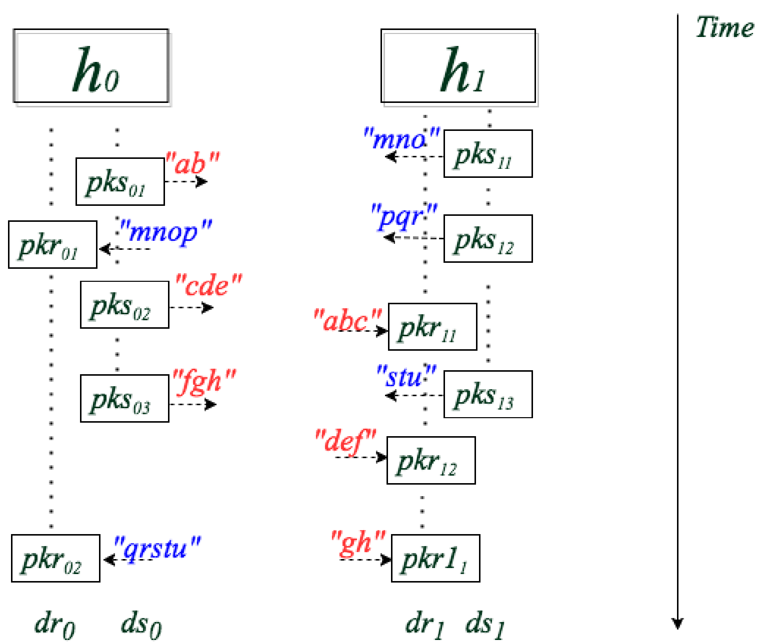

Figure 3.5: Example of Reliable Communication

In this example, the payloads of the packages are:

$pl(pks_01) = $ "ab", $pl(pks_02) = $ "cde", $pl(pks_03) = $ "fgh";

$pl(pkr_11) = $ "abc", $pl(pkr_12) = $ "def", $pl(pkr_13) = $ "gh" .

and

$pl(pks_11) = $ "mno", $pl(pks_12) = $ "pqr", $pl(pks_13) = $ "stu";

$pl(pkr_01) = $ "$mnop$", $pl(pkr_02) = $ "$qrstu$".

on the other direction. Their properties:

$pl(pks_01) \cdot pl(pks_02) \cdot pl(pks_03) = pl(pkr_11) \cdot pl(pkr_12) \cdot pl(pkr_13) = $ "$abcdefgh$" and

$pl(pks_11) \cdot pl(pks_12) \cdot pl(pks_13) = pl(pkr_01) \cdot pl(pkr_02) = $ "$mnopqrstu$".

satisfy the content preservation.

The relative time relationship of the packages are:

$time(pks_01) < time(pks_02) < time(pkr_11) < time(pks_03) < time(pkr_12) < time(pkr_13)$;

$time(pks_11) < time(pks_12) < time(pkr_01) < time(pks_13) < time(pkr_02)$.

The fact that

$pl(pkr_01) = $ "$mnop$" is the prefix of $pl(pks_11) \cdot pl(pks_12) = $ "$mnopqr$",

$pl(pkr_01) \cdot pl(pkr_02) = $ "$mnopqrstu$" is the prefix of(is this case identical to) $pl(pks_11) \cdot pl(pks_12) \cdot pl(pks_13) = $ "$mnopqrstu$",

$pl(pkr_11) = $ "abc" is the prefix of $pl(pks_01 \cdot pl(pks_02) = $ "$abcde$",

$pl(pkr_11) \cdot pl(pkr_12) = $ "$abcdef$" and $pl(pkr_11) \cdot pl(pkr_12) \cdot pl(pkr_13) = $ "$abcdefgh$" are the prefix of $pl(pks_01) \cdot pl(pks_02) \cdot pl(pks_03) = $ "$abcdefgh$"

satisfy the timing preservation.

Figure3.6 is an example of the unreliable communication.

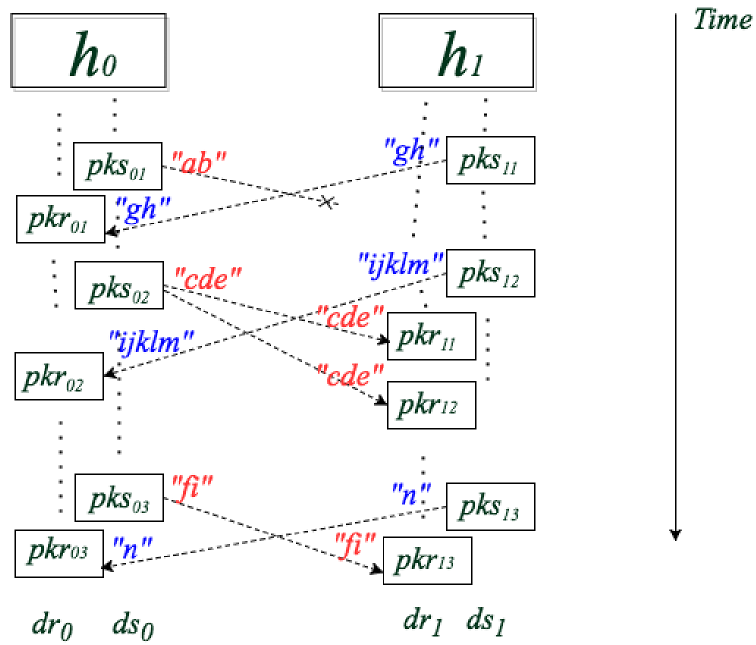

Figure 3.6: Example of Unreliable Communication

In this example:

$pkr_1 1 = pks_0 2 =$ "cde", $time(pkr_1 1) > time(pks_0 2)$;

$pkr_1 2 = pks_0 2 =$ "cde", $time(pkr_1 2) > time(pks_0 2)$;

$pkr_1 3 = pks_0 3 =$ "fi", $time(pkr_1 3) > time(pks_0 3)$;

$pkr_0 1 = pks_1 1 =$ "gh", $time(pkr_0 1) > time(pks_1 1)$;

$pkr_0 2 = pks_1 2 =$ "$ijklm$", $time(pkr_0 2) > time(pks_1 2)$;

$pkr_0 3 = pks_1 3 =$ "n", $time(pkr_0 3) > time(pks_1 3)$.

All of these satisfy the content preservation and timing preservation of the unreliable communication.

3.3 Dual-Trace Model

Before the modeling, I describe the facts of the dual-trace being analyzed. The traces in a dual-trace are in assembly level. One dual-trace contains two execution traces. There is no timing information of these two traces which means we don't know the time-stamps of the events of these two traces and can not match the events from both sides by time sequence. However the captured instructions in the trace are ordered in execution sequence. The execution traces contain all executed instructions as well as the corresponding changed memory by each instruction. Additionally, system calls are also captured by instruction id, which means if .dll or .exe files are provided, the system function calls can be identified with function names. Memory states can be reconstructed from the recorded memory changes to get the data information of the communication.

In this model, a dual_trace is defined as $dtr = \{tr0, tr1\}$

where $tr0$ and $tr1$ are two assembly execution traces of two interacting programs.

A trace is a sequence of executed instruction line. Hence, we can define a trace tr as a sequence of n lines:

$tr = (l_1, l_2, ..., l_n)$

Each instruction line has three attributes:

- *Memory changes:* each assembly instruction has operands which can be register or memory. These operands are being manipulated by the instruction and change the values they hold. The memory changes include all manipulated registers and memories as well as the changed values. we will use the notation

  $mem(l)$

  to denote this memory changes.

- *Function information:* an assembly instruction line might be an entry or return of a function. If it is an entry of a function, it would be able to acquire the function name with presence of the corresponding dynamic libraries. We assume that the function related information can be retrieved by some methods and stored along with the instruction line.

  $fun(l)$

  to denote this function information.

- *Line number:* each instruction in a trace is line. The line number of a instruction is the offset of it compare to the first instruction whose line number is 0.

  $num(l)$

  to denote this line number.

From the point of view of the communication analysis, the communications can be recovered by analysis of the relevant function calls(i.e. get the function name, function call line, function return line, input parameters and output parameters) in the traces. This function call can be treated as an event ev and defined as a tripled:

$ev =< funN, sl, el, in, out, type >$

where $funN$ is the function name, sl is the function call line, el is the function return line, in is the input parameters, out is the output parameters and type is the event type which can be one of the four types: channel open, data send, data receive and channel close.

After the definition of event, the trace can be represented in a sequence of events while ignoring all other unconcerned information. This new form of trace is called event trace and defined as etr :

$etr = (ev_1, ev_2, ..., ev_m)$

This event trace is acquired through the processing of the original trace and screen out the concerned function calls. The process to acquire function calls and filter out the relevant ones can be denoted by the notation:

$etr = eventfilter(tr)$

The events in etr belong to various data streams. The original trace can be also represented as stream trace str, which is a set of stream:

$str = \{s_1, s_2, ..., s_p\}$

where s_i is a stream consist of four sub string: channel open stream, data send stream, data receive stream and channel close stream and can be denoted as a tripled:

$s =< so, ss, sr, sc >$

The sub string sx in s consist of a sequence of events which is the sub sequence of etr.

$s =< ev_1, ev_2, ..., ev_q >$

Note 1: the event numbering here is different from the numbering in the etr definition, in another word, ev_1 in s is not the same event as ev_1 in etr.

Note 2: the events belong to the same sub stream has the same event type.

A process is used to further distinct them for data streams (i.e. data stream received and the data stream sent) of the corresponding communications, we use the notation:

$str = streamfilter(etr)$

to denote this process.

3.4 Relationship between Communication Model and Dual-Trace Model

The identification of the communication from dual_trace can be simply abstracted as finding the elements of each communication as defined in the communication model from the dual_trace.

A communication is defined as $c =< ch, e0, e1 >$ while a dual_trace is defined as $dtr = \{tr0, tr1\}$. In the dual_trace model, a trace tr can also be represented as stream trace $str = \{s_1, s_2, ..., s_p\}$. In the communication model, $e =< handle, d_r, d_s >$.

Each stream in str contains four sub stream: so, ss, sr, sc. The $handle$ of e and ch in c can be acquired from the events in so. d_r can be obtain from sr while d_s can be obtained from ss. And pkg in the data stream in the communication model has a one to one relationship with ev in the data send and receive stream in the dual_trace model.

By understanding this relationship, I am optimistic that as long as I can retrieve all the elements defined in the trace in dual_trace model, there will be a way to identify the communication. In next chapter algorithms for communication identification will be discussed in detail.

Chapter 4

Communication Identification Algorithms

This chapter discuss the algorithms for communication identification from dual-trace. Pseudo code are listed for algorithms. The algorithm is based on the models developed in the models Chapter3.

4.1 Communication Methods' Implementation in Windows

This section investigate the characteristics and the implementation of the communication methods. The goal of this investigation is to 1) obtain the system function set $fset$ for the concerned events in the communication and summarize the necessary parameters for further communication identification. and 2) understand the channel opening mechanism in order to identify the streams from the etr and match the streams from two traces.

The implementations of four communication methods in Windows system are investigated. I reviewed the Windows APIs of the communication methods and their example code. For each communication method, a system function list is provided for reference. These lists contain function names, essential parameters. These functions are supported in most Windows operating systems, such as Windows 8, Window 7. The channel opening mechanisms of each method are described in detail and represented in diagrams.

Windows API set is very sophisticated and multiple solutions are provided to fulfil a communication method. It is impossible to enumerate all solutions for each communication method. I only give the most basic usage provided in Windows documentation. Therefore, the provided system function lists for the events should not be considered as the only combination or solution for each communication method. With the understanding of the model, it should be fairly easy to draw out lists for other solutions or other communication methods.

Moreover, the instances of this model only demonstrate Windows C++ APIs. This model may be generalizable to other operating systems with the effort of understanding the APIs of those operating systems.

4.1.1 Windows Calling Convention

The Windows calling convention is important to know in this research. The communication identification relies not only on the system function names but also the key parameter values. In the assembly level execution traces, the parameter values is captured in the memory changes of the instructions. The memory changes are recognized by the register names or the memory address. The calling convention helps us to understand where the parameters are stored so that we can find them in the memory change map in the trace. Calling Convention is different for operating systems and the programming language. The Microsoft* x64 example calling convention is listed in B since we used dual-trace from Microsoft* x64 for case study in this work.

4.1.2 Named Pipes

In Windows, a named pipe is a communication method for the pipe server and one or more pipe clients. The pipe has a name, can be one-way or duplex. Both the server and clients can read or write into the pipe.[16] In this work, I only consider one server versus one client communication. One server to multiple clients scenario can always be divided into multiple server and client communications thanks to the characteristic that each client and server communication has a separate conduit. The server and client are endpoints in the communication. We call the server "server endpoint" while the client "client endpoint". The server endpoint and client endpoint of a named pipe share the same pipe name, but each endpoint has its own buffers and handles.

There are two modes for data transfer in the named pipe communication method, synchronous and asynchronous. Modes affect the functions used to complete the send and receive operation. I list the related functions for both synchronous mode and asynchronous mode. The create channel functions for both modes are the same but with different input parameter value. The functions for send and receive message are also the same for both cases. However, the operation of the send and receive functions are different for different modes. In addition, an extra function *GetOverlappedResult* is being called to check if the sending or receiving operation finish, the output message will be stored in the overlap structure whose memory address saved in the function's output parameter Overlap Structure Address. Table4.1 lists the functions of the events for synchronous mode while Table4.2 lists the functions of the events for the asynchronous mode for a Named pipe communication.

Table 4.1: Function List of events for Synchronous Named Pipe

Event	Server Endpoint		Client Endpoint	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	CreateNamedPipe	RAX: File Handler	CreateFile	RAX: File Handler
		RCX: File Name		RCX: File Name
Channel Open	ConnectNamedPipe	RCX: File Handler		
Send	WriteFile	RCX: File Handle	WriteFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Receive	ReadFile	RCX: File Handle	ReadFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Channel Close	CloseHandle	RCX: File Handler	CloseHandle	RCX: File Handler
Channel Close	DisconnectNamedPipe	RCX: File Handler	DisconnectNamedPipe	RCX: File Handler

Table 4.2: Function List of events for Asynchronous Named Pipe

Event	Server Endpoint		Client Endpoint	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	CreateNamedPipe	RAX: File Handler	CreateFile	RAX: File Handle
		RCX: File Name		RCX: File Name
Channel Open	ConnectNamedPipe	RCX: File Handler		
Send	WriteFile	RCX: File Handle	WriteFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Receive	ReadFile	RAX: File Handle	ReadFile	RCX: File Handle
		RDX: Buffer Address		RDX: Buffer Address
		R9: Message Length		R9: Message Length
Receive	GetOverlapped-Result	RCX: File Handler	GetOverlapped-Result	RCX: File Handler
		RDX: Overlap Structure address		RDX: Overlap Structure Address
Channel Close	CloseHandle	RCX: File Handler	CloseHandle	RCX: File Handler
Channel Close	DisconnectNamedPipe	RCX: File Handler	DisconnectNamedPipe	RCX: File Handler

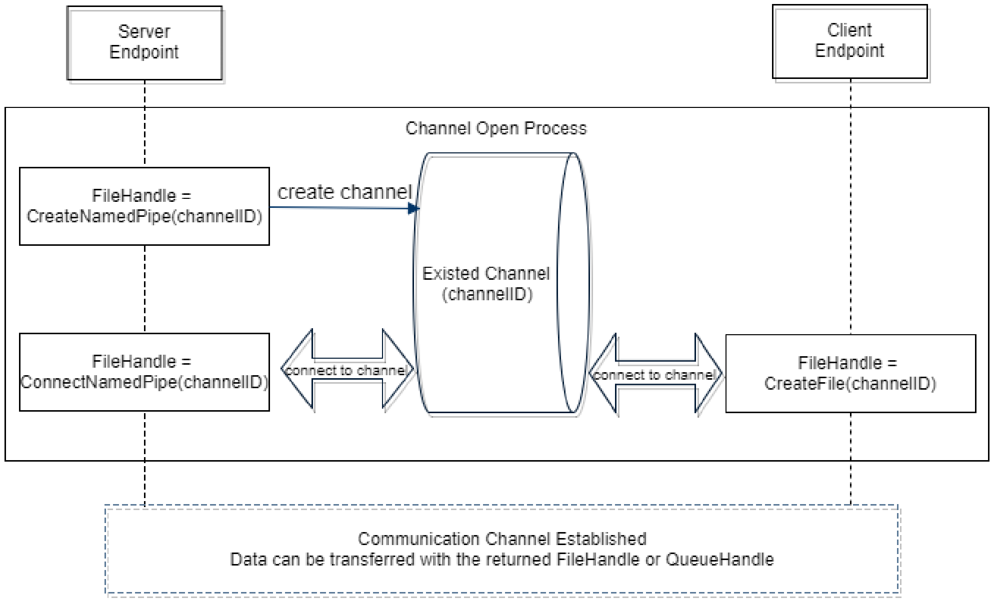

Figure 4.1: Channel Open Process for a Named Pipe

A named pipe server is responsible for the creation of the pipe, while clients can connect to the pipe after it was created. The creation and connection of a named pipe returns the handle ID of that pipe. These handler Ids will be used later when data is being sent or received to a specified pipe. Figure4.1 shows the channel set up process for a Named Pipe communication.

4.1.3 Message Queue

Similar to Named Pipe, Message Queue's implementation in Windows also has two modes, synchronous and asynchronous. Moreover, the asynchronous mode further divides into two operations, one with callback function while the other without. With the callback function, the callback function would be called when the send or receive operations finish. Without callback function, the general function *MQGetOverlappedResult* should be called by the endpoints to check if the message sending or receiving operation finish, the output message will be stored in the overlap structure whose memory address saved in the function's output parameter Overlap Structure Address. Table4.3 lists the functions for synchronous mode while Table4.4 and Table4.5 list the functions for the asynchronous mode with and without callback.

Table 4.3: Function List of events for Synchronous MSMQ

Event	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Send	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
Receive	MQReceiveMessage	RCX: Queue Handle
		R9: Message description structure Address
Channel Close	MQCloseQueue	RCX: Queue Handler

Table 4.4: Function List of events for Asynchronous MSMQ with Callback

Event	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Send	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
Receive	MQReceiveMessage	RCX: Queue Handle
		R9: Message description structure Address
Receive	CallbackFuncName	Parameters for the callback function.
Channel Close	MQCloseQueue	RCX: Queue Handler

Table 4.5: Function List of events for Asynchronous MSMQ without Callback

Event	**Function**	**Parameters**
Channel Open	MQOpenQueue	RAX: Queue Handler
		RCX: Queue Format Name
Send	MQSendMessage	RCX: Queue Handle
		RDX: Message description structure Address
Receive	MQReceiveMessage	RCX: Queue Handle
		R9: Message description structure Address
Receive	MQGetOverlappedResult	RCX: Overlap Structure address
Channel Close	MQCloseQueue	RCX: Queue Handler

The endpoints of the communication can create the queue or use the existing one. However, both of them have to open the queue before they access it. The handle ID returned by the open queue function will be used later on when messages are being sent or received to identify the queue. Figure4.2 shows the channel set up process for a Message Queue communication.

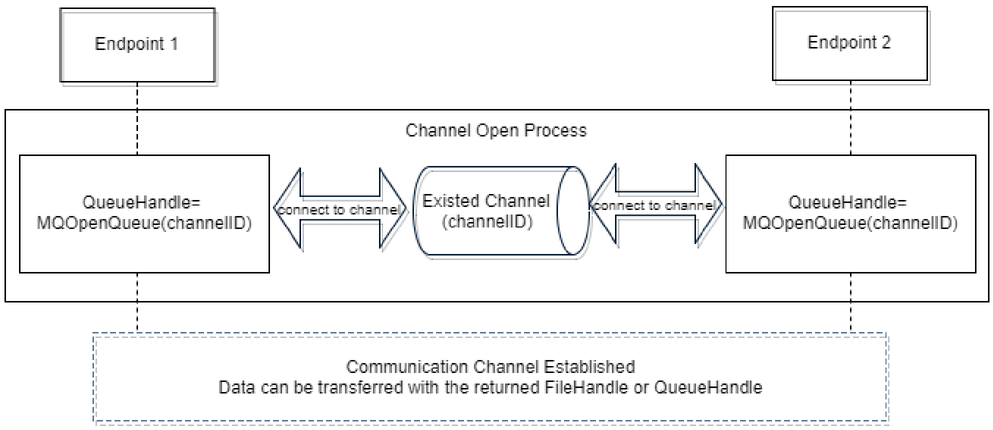

Figure 4.2: Channel Open Process for a Message Queue

4.1.4 TCP and UDP

In Windows programming, these two methods shared the same set of APIs regardless the input parameter values and operation behaviour are different. In Windows socket solution, one of the two endpoints is the server while the other one is the client. Table 4.6 lists the functions of a UDP or TCP communication.

Table 4.6: Function List of events for TCP and UDP

Event	Server Endpoint		Client Endpoint	
	Function	**Parameters**	**Function**	**Parameters**
Channel Open	socket	RAX: Socket Handle	socket	RAX: Socket Handle
Channel Open	bind	RCX: Socket Handle	connect	RCX: Socket Handle
		RDX: Server Address & Port		RDX: Server Address & Port
Channel Open	accept	RAX: New Socket Handle		
		RCX: Socket Handle		
		RDX: Client Address & Port		
Send	send	RCX: New Socket Handle	send	RCX: Socket Handle
		RDX: Buffer Address		RDX: Buffer Address
Receive	recv	RCX: New Socket Handle	recv	RCX: Socket Handle
		RDX: Buffer Address		RDX: Buffer Address
Channel Close	closesocket	RCX: New Socket Handle	closesocket	RCX: Socket Handle

The communication channel is set up by both of the endpoints. The function *socket* should be called to create their own socket on both endpoints. After the sockets are created, the server endpoint binds the socket to its service address and port by calling the function *bind*. Then the server endpoint calls the function *accept* to accept the client connection. The client will call the function *connect* to connect to the server. When the function *accept* return successfully, a new socket handle will be generated and returned for further data transfer between the server endpoint and the connected client endpoint. After all these operations are performed successfully, the channel is established and the data transfer can start. During the channel open stage, server endpoint has two socket handles, the first one is used to listen to the connection from the client, while the second one is created for real data transfer. Figure4.3 shows the channel open process for TCP and UDP.

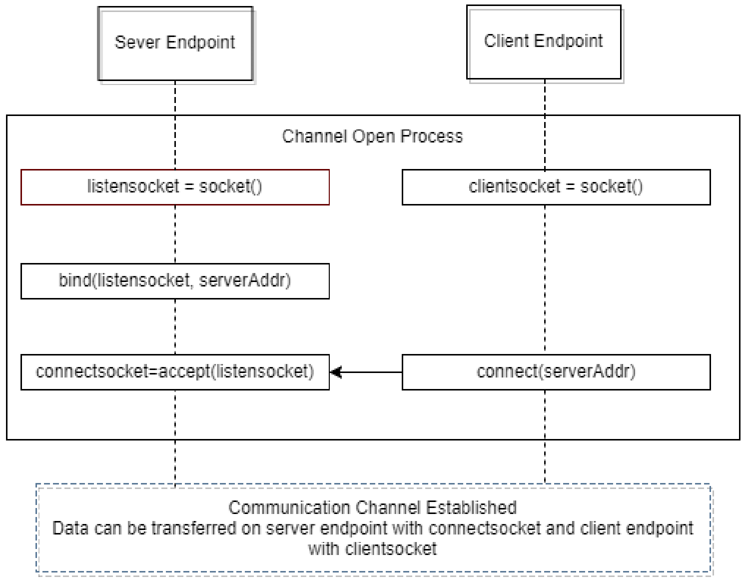

Figure 4.3: Channel Open Model for TCP and UDP

4.2 Communication Identification Algorithm

The identification of the communications from a dual_trace should be able to identify the concerned communications as well as all the components defined in it. The inputs of this algorithm are the $dual\_trace = \{tr0, tr1\}$ and the concerned communication method's function set $fset = \{f_1, f_2...f_m\}$. The output of this algorithm is all the identified communications of the concerned communication method. This is a very high level algorithm, details of each step in this algorithm will be discussed in the later sections.

Algorithm 1: Communication Identification Algorithm

Input: $dual\_trace$, $fset$

Output: $cs = \{c_1, c_2...c_n\}$

$i = 0$;

for $tr \in dual\_trace$ **do**

  $etri = eventfilter\,(tr, fset)$;

  $stri = streamfilter\,(etr)$;

$cs = streammatch(str0, str1)$;

return cs;

4.3 Event Locating Algorithm: $eventfilter\left(\right)$

The concerned events in a communication are channel open, channel close, send and receive events. These events are identified as system function calls in this work. A function call in the trace starts from the function call instruction to the function return instruction. The input parameters' value and input buffer content should be retrieved from the memory state of the the function call instruction line while the return value, output parameters' value and output buffer content should be retrieved from the memory state of the function return instruction line. Tables in section 4.1 indicate all the functions of the communication methods as well as the concerned parameters. Following the windows calling convention, the concerned parameter value or buffer address can be found in the corresponding register or stack positions. The buffer content can be found in the memory address in the reconstructed memory state. Each event can be completed by different function calls. For example, for the client endpoint in TCP communication method, both *socket* and *connect* function call are considered to be the channel open events. The functions list for a communication method is needed as a input of this algorithm. Tables in Section4.1 give the examples of function list of the events for some communication methods. The algorithm presented in this section is designed for locating all function calls provided in the function list as events of one communications method. If more than one communication methods are being investigated, this algorithm should be run multiple times, each for a method. Events in the output event list is sorted by time of occurrence. Since the function list usually contain a very small number of functions compared to the instruction line number in the execution trace, the time complexity of this algorithm is O(N+M) , N and M are the instruction line numbers of the two traces in the duel-trace.

Algorithm 2: Event Locating Algorithm

Input: $tr, fset$

Output: etr

etr ← empty event list;
while *not at end of trace* **do**
  for $f \in fset$ **do**
    if *Is function call of f* **then**
      $ev.funN = f.funN$ $ev.startline$ ← current Line number;
      $ev.endline$ ← find function return instruction line;
      $ev.in$ ← reconstruct memory of $event.startline$ from the trace and get input values of $f.pars$;
      $ev.out$ ← reconstruct memory of $event.endline$ from the trace and get outputs values of $f.pars$;
      $ev.type$ ← $f.type$;
      $success$ ← get the success code from $event.out$;
      if $success$ **then**
        $etr.add(ev)$;
return etr;

4.4 Stream Identification Algorithm: $streamfilter()$

The events located in the etr may belong to different stream, the next step in the communication identification algorithm is to identify them for each stream. The input of this algorithm the etr from the "Event Locating Algorithm". Since the input etr is sorted by time of occurrence and the channel open events should always happen before other events, it is reasonable to assume the new stream can be identified by its first channel open function call. The identification for TCP and UDP server endpoints are slightly complicated than the other ones, due to its own channel open mechanism. The output of this algorithm is the str. Each stream in this str consist of the sub streams. The concepts of the stream and sub streams are defined in SectionA.

Algorithm 3: Stream Indentification Algorithm

Input: etr

Output: str

$str \leftarrow Map\langle h, s\rangle$;

for $ev \in etr$ **do**

  if ev *is a channel open event* **then**

    $h \leftarrow$ get the handle identifier from the function parameter list;

    $s \leftarrow str.get\,(handle)$;

    if ev *is an* $accept\,()$ *function call for TCP or UDP* **then**

      $h1 \leftarrow$ get the second socket handle identifier from $ev.out$;

      $str.remove\,(h)$;

      $str.add\,(h1, s)$;

    if s *is null* **then**

      New a stream s;

      $str.add\,(h, s)$;

    $ss.so.add\,(evs)$;

  if ev *is a channel send event* **then**

    $h \leftarrow$ get the handle from the function parameter list;

    $s \leftarrow str.get\,(h)$;

    if s *is not* $null$ *and* $s.complete$ *is* $False$ **then**

      $s.ss.add\,(ev)$;

  if ev *is a channel receive event* **then**

    $h \leftarrow$ get the handle from the function parameter list;

    $s \leftarrow str.get\,(h)$;

    if s *is not* $null$ *and* $s.complete$ *is* $False$ **then**

      $s.sr.add\,(event)$;

  if ev *is a channel close event* **then**

    $h \leftarrow$ get the handle from the function parameter list;

    $s \leftarrow str.get\,(h)$;

    if s *is not null* **then**

      $s.sc.add\,(ev)$;

      $s \leftarrow True$;

return str;

4.5 Stream Matching Algorithm: $streammatch()$

The communication identification algorithm aims at identifying all the communication of a concerned communication method from the dual-trace. The input of this algorithm is the two str from the dual-trace. The output of this algorithm is the communication list. Each communication recognized from the dual_trace contains two streams. The channel of a communication defined in Section3.2 is not explicitly represented in the output but it was implicitly used in this algorithm.

In the communication identification algorithm, it first try to match two streams to a channel only by their identifiers. In this level, the matching depends on channel open mechanisms which are different from communication method to communication method. For TCP and UDP the matching can be considered as local address and port of server endpoint matching with remote address and port of client endpoint. For Named Pipe, it uses the file name, while for Message Queue, it uses the queue name as the identifier for matching of two endpoints.

The first level matching can not guarantee the exact endpoints matching and channel identification. There are two situations which false negative error might emerge. Take Named Pipe for example, the first situation is multiple(more than two) interacting programs shared the same file or queue as their own channel. Even though the channels are distinct for each communication, but the file or queue used is the same one. For example, the Named Pipe server is connected by two clients using the same file. In the server trace, there are two streams found. In each client trace, there is one stream found. For the dual_trace of server and client1, there will be two possible identified communications, one is the real communication for server and client1 while the other is the false negative error actually is for server and client2. The stream in client1's trace will be matched by two streams in the server's trace. The second situation is the same channel is reused by the different endpoints in the same programs. For example, the Named Pipe server and client finished the first communication and then closed the channel. After a while they re-open the same file again for another communication. Since the first level matching is only base on the identifiers and the first and the second communications have the same identifier since they used the same file. Similar situations can also happen in Message Queue, TCP and UDP communication methods.

To reduce the false negative error, the second level matching should be applied, which is also being named as transmitted data verification algorithm. On top of the endpoint identifiers matching, further data verification should be applied to make sure the matching is reliable. This verification crossly compare the sent and received data in both streams in the first level matching. If the transmitted data in the streams are considered to be identical, the matching is confirmed, otherwise it was a false negative error. However, we still can not exclude all the false negative errors, due to the data transmitted in two communication can be identical. Figure4.4 indicates the ineffective second level matching scenario and the effective one.

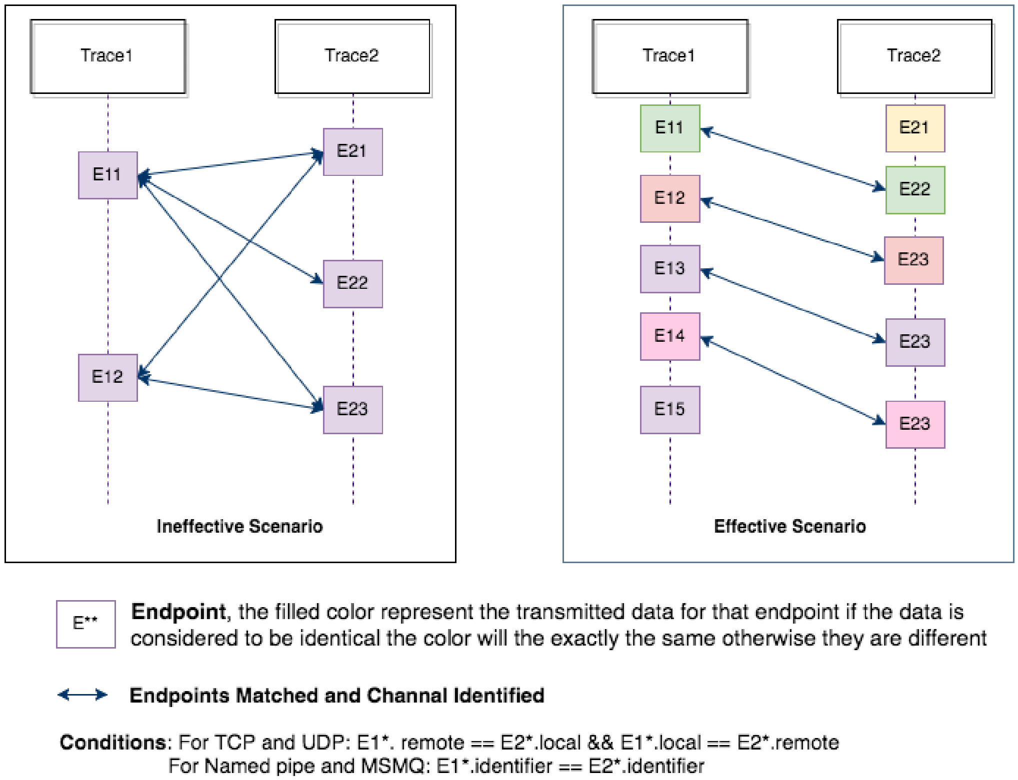

Figure 4.4: Second Level Matching Scenarios

The following subsections discuss the algorithms for these two level matching. In Section4.1, I elaborate the channel open process and the data transfer categories for the concerned communication methods. Based on the different channel opening process, two algorithms are developed for the communication identification, one is for Named Pipe and Message Queue, the other is for TCP and UDP. The inputs of the these two algorithms are the same, two $strs$ from the original dual_trace.

The data transfer characteristics divided the communication methods into reliable and unreliable transmissions. Named Pipe and TCP fall in the reliable category while Message Queue and UDP fall in the unreliable one. The second level matching algorithms are different for these two categories. The corresponding second level data verification algorithms are being used in the communication identification algorithms. The inputs of the transmitted data verification algorithms are streams matched in the first level matching while the output a boolean to indicate if the transmitted data of this two streams are matched and the verified data.

4.5.1 Stream Matching Algorithm for Named Pipe and Message Queue

For Named Pipe and Message Queue, only one channel open function is being called in each s. So in the below algorithm, when it try to get the channel open event from the $s.so$ list, only one event should be found and return. The channel identifier parameters can be found in the $ev.in$ of the channel open event. The identifier for Named Pipe is the file name of the pipe while for Message Queue is the format queue name of the queue. This algorithm finds out all the possible communications regardless some of them might be false negative errors.

Algorithm 4: Stream Matching Algorithm for Named Pipe and Message Queue

```
Input: str0, str1
Output: cs = {c_1, c_2...c_n}
cs ← empty communication list;
for s0 ∈ str0 do
    openev0 ← get the opening event from s0.so;
    chId0 ← get the channel identifier from openev0.in;
    for s1 ∈ str1 do
        openev1 ← get the opening event from s1.so;
        chId1 ← get the channel identifier from openev1.in;
        if chId0 == chId1 then
            DataVerified = dataVerify(s0, s1, outputdata);
            if DataVerified == True then
                c.stream0 = stream0;
                c.stream1 = stream1;
                c.dataMatch = outputdata;
                cs.add (c);
return cs;
```

4.5.2 Stream Matching Algorithm for TCP and UDP

For TCP and UDP multiple functions are collaborating to create the final communication channel. The local address and port of the server endpoint and the remote address and port of the client endpoint are used to identify the channel. This algorithm first try to retrieve the local address and port of the server endpoint and remote address and port from client endpoint. Then it try to match two endpoints by comparing the local and remote address and port. Transmitted data verification also applied in this algorithm.

Algorithm 5: Stream Matching Algorithm for TCP and UDP

Input: $str0, str1$
Output: $cs = \{c_1, c_2 ... c_n\}$

$cs \leftarrow$ empty communication list;
for $s0 \in str0$ **do**
    $socketev0 \leftarrow$ get the $socket\,()$ event from $str0.so$;
    $bindev0 \leftarrow$ get the $bind\,()$ event from $str0.so$;
    $connectev0 \leftarrow$ get the $connect\,()$ event from $str0.so$;
    for $s1 \in str1$ **do**
        $socketev1 \leftarrow$ get the $socket\,()$ event from $s1.so$;
        $bindev1 \leftarrow$ get the $bind\,()$ event from $s1.so$;
        $connectev1 \leftarrow$ get the $connect\,()$ event from $s1.so$;
        if $socketev0! = null$ *AND* $socketev! = null$ **then**
            if $bindev0! = null$ *AND* $connectev1 == null$ **then**
                $localServerAddr \leftarrow$ get serverAddr from $bindev1.in$;
            else if $bindev1 == null$ *AND* $connectev0! = null$ **then**
                $remoteServerAddr \leftarrow$ get serverAddr from $connectev1.in$;
            else
                Break the inner For loop;
            if $localServerAddr == remoteServerAddr$ **then**
                $DataVerified = dataVerify(stream0, stream1, outputdata)$. **if** $DataVerified == True$ **then**
                    $c.s0 = s0$;
                    $c.s1 = s1$;
                    $c.dataMatch = outputdata$; $cs.add\,(c)$;
return cs;

4.5.3 Data Verification $dataVerify()$ for Named Pipe and TCP

As described in Section3.1.1, the data being received by one endpoint should always equal to or at least is sub string of the data being sent from the other endpoint in a communication for the reliable transmission methods, such as Named Pipe and TCP. So the data verification algorithm is in data union level. The send data union is retrieved by the concatenation of the input buffer content of the send events in the send stream of an endpoint. The receive data union is retrieved by the concatenation of the output buffer content of the receive events in the receive stream of the other endpoint. The input of this algorithm is the two $streams$ from two traces which are being matched in the first level.

Algorithm 6: Transmitted Verification by Data Union

Input: $s0, s1$

Output: send data union and receive data union of two streams

return *Indicator of if transmitted data union are considered to be identical*
$send1 \leftarrow$ empty string;
$send2 \leftarrow$ empty string;
$recv1 \leftarrow$ empty string;
$recv2 \leftarrow$ empty string;
for $sendEvent \in s0.ss$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.in$;
    $send0.append\,(sendmessage)$;
for $sendEvent \in s1.ss$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.in$;
    $send1.append\,(sendmessage)$;
for $recvEvent \in s0.sr$ **do**
    $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.out$;
    $recv0.append\,(sendmessage)$;
for $recvEvent \in s1.sr$ **do**
    $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.out$;
    $recv1.append\,(sendmessage)$;
if $recv0$ *is substring of* $send1$ *AND* $recv1$ *is substring of* $send0$ **then**
    return True;
else
    return False;

4.5.4 Data Verification $dataVerify()$ for MSMQ and UDP

For the unreliable communication methods, the data packets being transmitted are not delivery and ordering guaranteed. So it is impossible to verify the transmitted data as a whole chunk. Fortunately, the packets arrived to the receivers are always as the original one from the sender. Therefore, we perform the transmitted data verification by single events instead of the whole stream. This algorithm basically goes through events of the ss in one stream trying to find the matched receive event in the sr in the other stream. And then calculate the fail packet arrival rate. The fail packet arrival rate should be comparable to the packet lost rate. So we set the packet lost rate as the threshold to determine if the transmitted data can considered to be identical in both directions. The packet lost rate can be various from network to network or even from time to time for the same network. The inputs of this algorithm are the copies of two streams from two traces which are being matched and the packet lost rate as the threshold. I use copies instead of original data is to modify the input list directly in the algorithm. The threshold should be an integer. For example if the lost rate is 5%, the threshold should be set as 5.

Algorithm 7: Transmitted Verification by Data of Events

Input: $s0, s1$

Output: matched event list of two endpoints

return *Indicator of if transmitted data union are considered to be identical*
$sendPktNum0 \leftarrow s0.ss.length$;
$sendPktNum1 \leftarrow s1.ss.length$;
$recvPktNum0 \leftarrow 0$;
$recvPktNum1 \leftarrow 0$;
$eventMatchs \leftarrow List\langle EventMatch \rangle$;
for $sendEvent \in s0.ss$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.in$;
    for $recvEvent \in s1.sr$ **do**
        $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.out$;
        if $sendmessage == recvmessage$ **then**
            $recvPktNum0 + +$;
            $stream1.sr.remove\,(recvEvent)$;
            $eventMatch = NeweventMatch\,()$;
            $eventMatchs.add\,(eventMatch)$;
if $(sendPktNum0 - recvPktNum0) * 100/sendPktNum0 > threshold$ **then**
    return False;
for $sendEvent \in s1.ss$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    for $recvEvent \in s0.sr$ **do**
        $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.out$;
        if $sendmessage == recvmessage$ **then**
            $recvPktNum1 + +$;
            $s0.sr.remove\,(recvEvent)$;
if $(sendPktNum1 - recvPktNum1) * 100/sendPktNum1 > threshold$ **then**
    return False;
return True;

Chapter 5

Feature Prototype On Atlantis

In this section, I describe the design of the feature prototype of communication identification from the dual_trace. This feature is implemented on Atlantis and is built on top of Atlantis' other features, such as "memory reconstruction", "function inspect" and "views synchronization". Atlantis is an assembly trace analysis environment. It provides many powerful and novel features to assist assembly level execution trace analysis.[11] This prototype implemented the algorithms described in Chapter4 as well as the user interfaces for the feature.

This prototype consist of four main components: 1) user defined setting for defining the concerned communication methods' function set. 2) a view that can parallelly present both traces in the dual_trace. 3) two identification features: Stream identification and communication identification. 4) functionality that allow user to access the identification result.

5.1 User Defined Function Set

As emphasized in Section4.1, the function set for each communication method can be different depends on the implementation solution of the method. Furthermore, there are so many communication methods in the real world and not all of them are being analyzed by the user. Instead of using hard coded function sets, a configuration file in Json format is used for the users to define their concerned communication methods and the corresponding function set. This function sets will be the input for the communication identification. All concerned communication methods have its own function set. The identification features implemented in this prototype iterate all methods in the Json configuration file named "communicationMethods.json" and identify all communications of each method. This configuration includes the communication method, their function set for the communication events and the essential parameters of each function. A default template is given for user reference, this default template is generated by Atlantis when it was launched and stored in the .tmp folder in the trace analysis project folder. The default template example can be find in SectionC.

5.2 Parallel Editor View For Dual_Trace

The dual_trace consist of two execution traces which are interacting with each other. Presenting them in the same view makes the analysis for the user much easier. The strategy to open parallel editor view is that open one trace as the normal one and the other as the dual_trace of the current opened one. A new menu option in the project navigation view are created to open the second trace as the dual_trace of the current active trace. The implementation of the parallel editor take the advantage of the existing SWT of Eclipse plug-in development. The detail of the implementation can be found in SectionD. Figure5.1 shows this menu option and FigureD shows the parallel editor view.

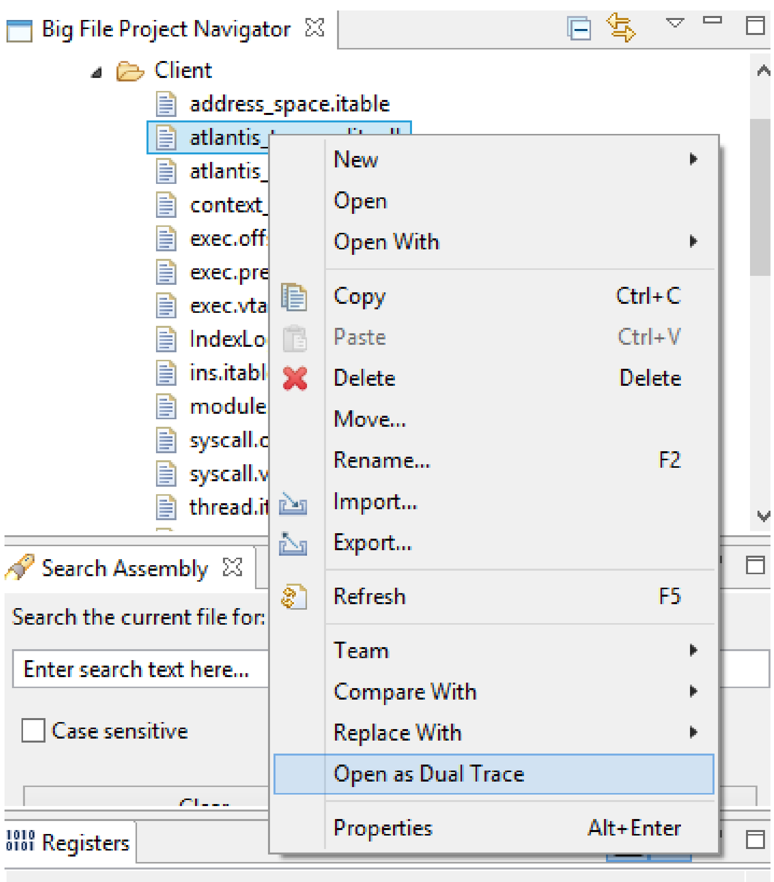

Figure 5.1: Menu Item for opening Dual_trace

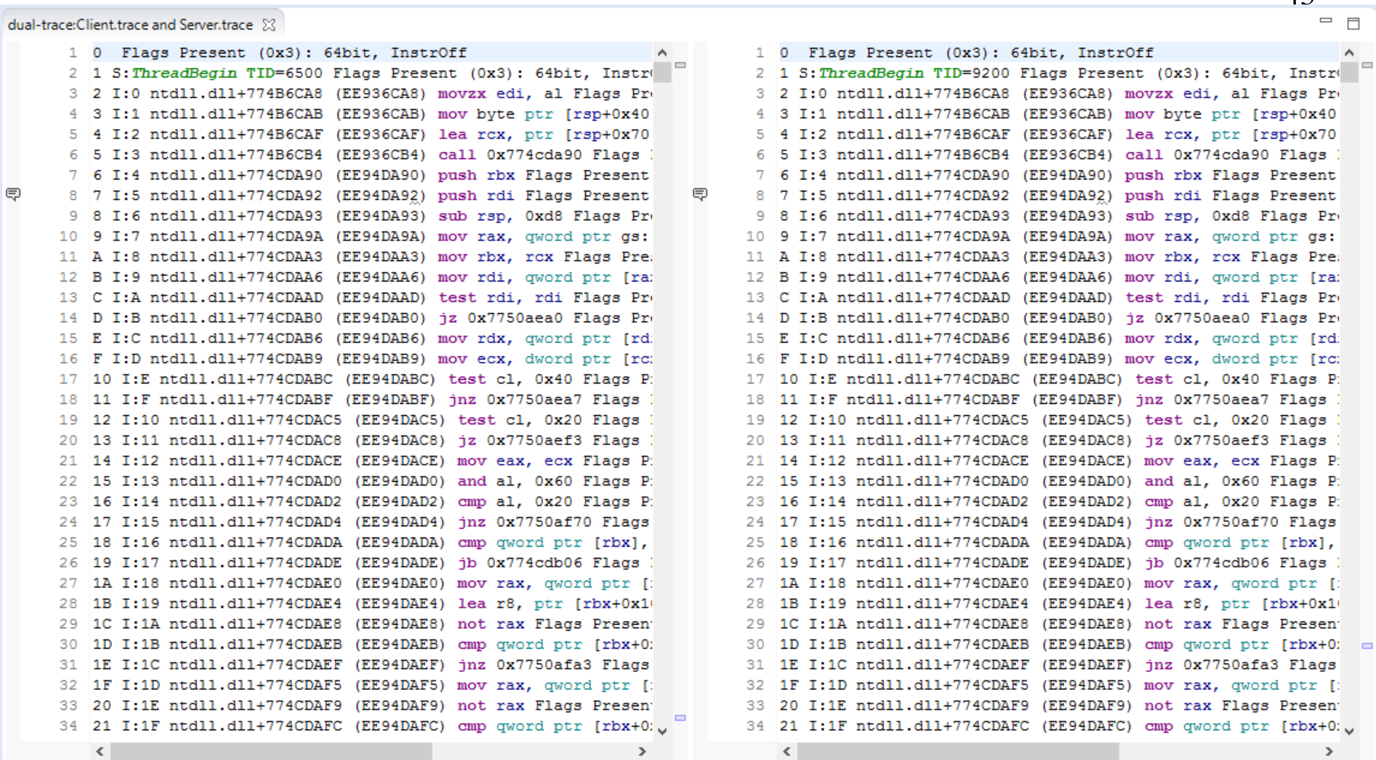

Figure 5.2: Parallel Editor View

5.3 Identification Features

I implemented two identification features, one is stream identification for both traces in the dual_trace, the other is the communication identification. These two features align to the "stream identification algorithm" and "communication identification algorithm" designed in Chapter4. The implementation of these two identification features relies on the existing "function inspect" feature of Atlantis. The called functions' name can be inspected by search of the symbolic name in the executable binary or any DLLs which used by the program at the time when it is traced. By importing the DLLs and executable binary, Atlantis can recognize the function call from the execution trace by the function names. Therefore the corresponding Dlls or executable binaries for both traces in the dual_trace have to be loaded into Atlantis before conducting the identification.

A new menu "Dual_trace Tool" with three menu options is designed for these two identification features. In this menu, two options are for conducting the identification which are "Stream Identification" and "Communication Identification" while one is for loading the DLLs and executable binary which is "Load Library Exports". Currently, the "Load library export" function can only load libraries for the trace in the active editor. So this item in the menu has to be run twice separately for each trace of the dual_trace. Figure5.3 shows this new menu in Atlantis. When the user perform any of the identification features, there is the prompt dialog as shown in Figure5.4 which asks the user what communication methods they want to identify from the dual_trace. This list is provided by the configuration file I mention in Section5.1. The user can select one or multiple methods.

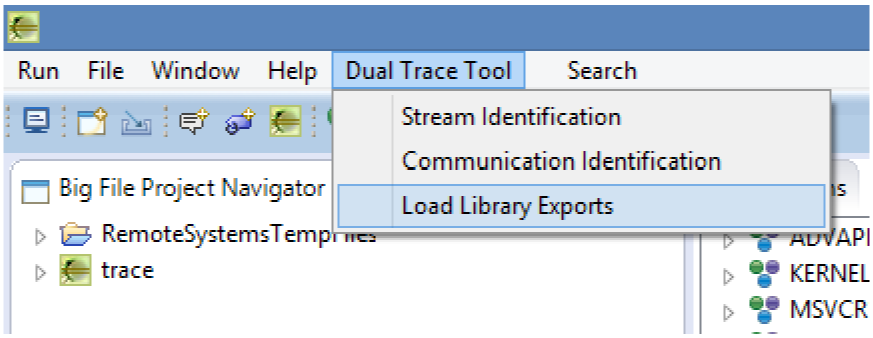

Figure 5.3: Dual_trace Tool Menu

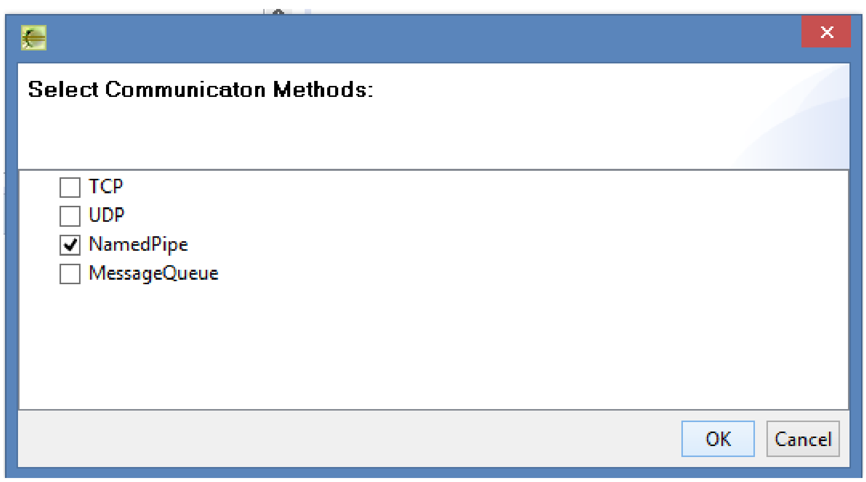

Figure 5.4: Prompt Dialog for Communication Selection

A new view named “Communication” is designed for presenting the result of the identification of streams and communications. Since the user can have multiple selection for communication methods they concern, the output identification result contains all the identified communications or streams of all the concerned communication methods and the identified results are clustered by methods. There are two sub tables in this view, the left one is for the stream identification result while the left one is for communication identification result. The reason for putting this two result in the same view is for easy access and comparison of the data for the users. Figure5.6 shows this view with result data in it. Each time when the user rerun the identification features the result in the corresponding table will be refreshed to show only the latest identification result. But the other table will not be affected. For example, if the user run the "Stream Identification" feature first, the stream identification result will show on the left table of the view. And then the user run the "communication Identification", the communication identification result will be shown on the right table while the left one still holding the last stream identification result.

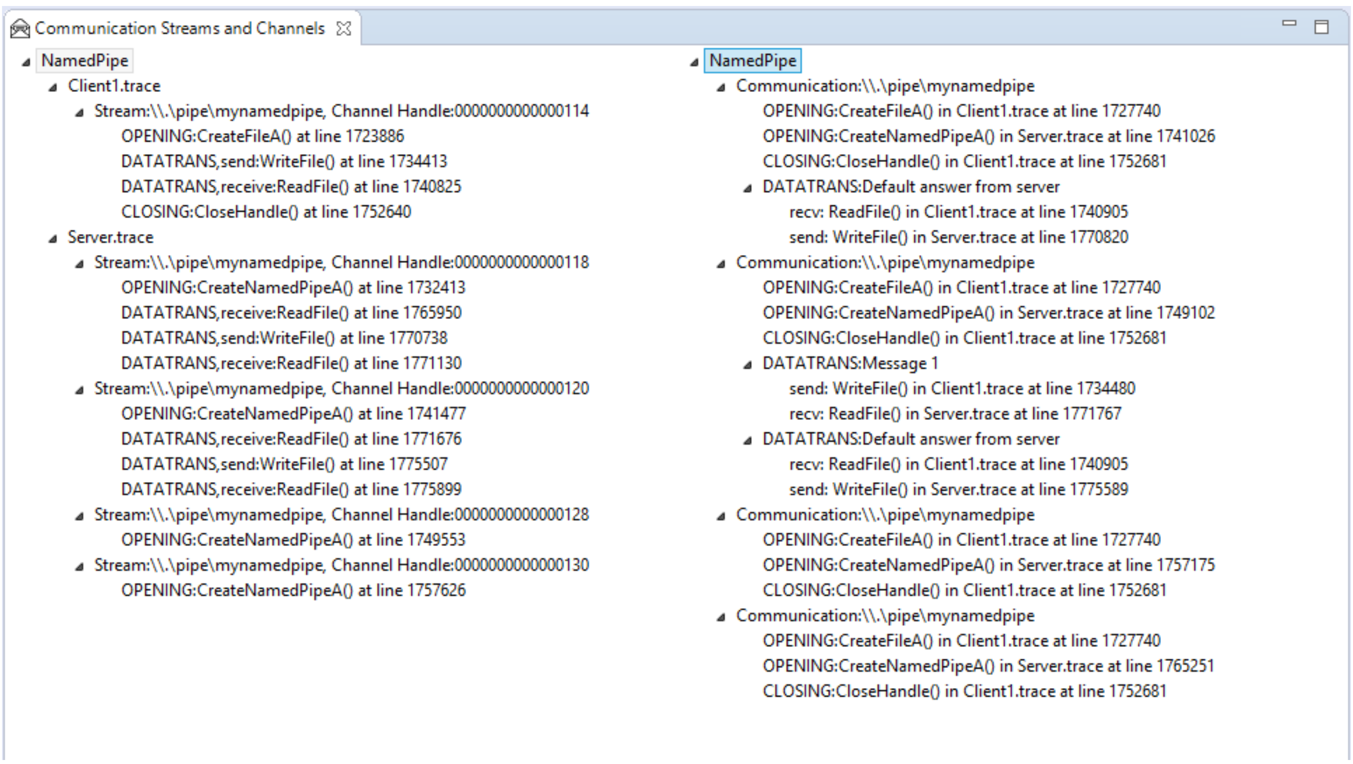

Figure 5.5: Communication View for Showing Identification Result

5.4 Identification Result View and Result Navigation

Atlantis is a analysis environment that has various views to allow user access to different information from the trace, such as the memory and register state of the current instruction line. Moreover, these views synchronize automatically with the editor view. These functionality and information also benefit the communication analysis of the dual_trace. Providing the user a way to navigate from the identified result to the traces in the editors allows them to take advantage of the current existing functionality of Atlantis and make their analysis of the dual_trace more efficient.

In the result list, each event entry is corresponding to a function call. The functions were called at function call line and all the inputs of the function calls can be recovered from the memory state of this instruction line. The functions returned at the return instruction lines, all the outputs of the function calls can be recovered in the memory state of the the return instruction line. From the event entries, this implementation provide two different ways for the user to navigate back to where the function begins and ends. When the user "double click" on an entry, it will bring the user to the start line of the function in the corresponding trace editor. When the the right click on the event entry, a prompted menu with the option "Go To Line of Function End" will show up as in Figure??. Clicking on this option will bring the user to the return line of this function in the trace editor. All other views update immediately with this navigation.

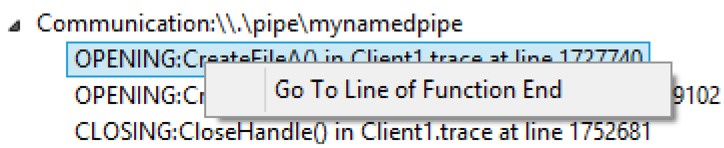

Figure 5.6: Right Click Menu on Event Entry

Moreover, the "remove" option as shown in Figure5.7 in the right click menu on the "stream or "communication" entries is provided for the user to remove the selected "stream" or "communication" entry. This provides the user the flexibility to get rid of the data that they don't care.

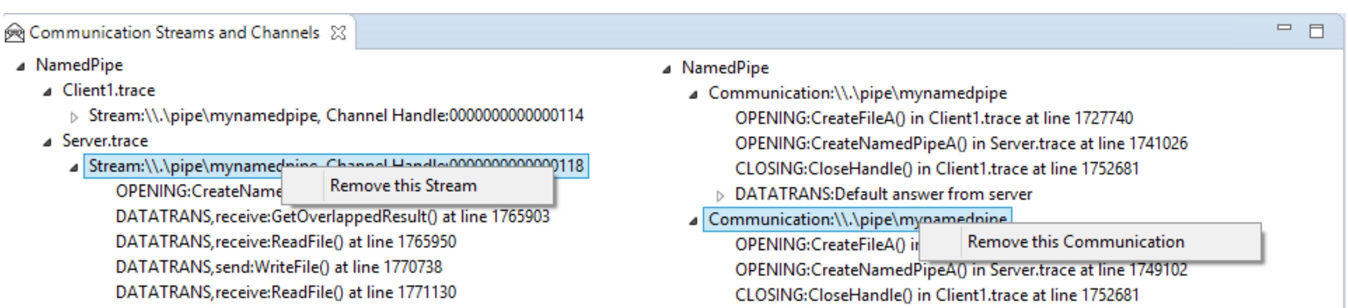

Figure 5.7: Right Click Menu on Event Entry

5.5 Data Structures for Identified Communications

The information of identified communications should be organized properly for the user. In this section, I define the output data structures to fulfil this requirement. There are totally two major data set. The first one is clustered as communications aligning the definition at Section3.2. The second one is clustered by endpoints in the traces. The reason to provide the second data set is due to the false negative errors of the channel identification. The identified endpoint lists of the traces provide more original data information. So with other assistant information and the access of this relatively original information of the dual-trace, the user has more flexibility to analysis the dual-trace. The data structures have been used in the algorithms implicitly.

Algorithm 8: Data Structure for Identified Communications

```
cs ← Map⟨String, List⟨Communication⟩⟩;    // the key is the communication method
str ← Map⟨String, List⟨Stream⟩⟩;    // the key is the communication method
struct {
    Stream s0                  // s0 is from tr0 of the dual-trace
    Stream s1                  // s1 is from tr1 of the dual-trace
    DataMatch dataMatch
} Communication
union {
    DataUnionMatch    unionMatch           // For data union verification
    List ⟨ EventMatch ⟩  eventMatchs       // For data event verification
} DataMatch
struct {
    String sData1              // send data union of endpoint1
    String rData1   // receive data union of endpoint1,substring of sData2
    String sData2              // send data union of endpoint2
    String rData2   // receive data union of endpoint2,substring of sData1
} DataUnionMatch
struct {
    Event     event1           // event1 is from enpoint1
    Event     event2           // event2 is from enpoint2
} EventMatch
struct {
    Int      handle
    List ⟨ Event ⟩  openStream
    List ⟨ Event ⟩  closeStream
    List ⟨ Event ⟩  sendStream
    List ⟨ Event ⟩  receiveStream
} Stream
struct {
    Int                      stratline
    Int                      endline
    Map ⟨ String, String ⟩   inputs
    Map ⟨ String, String ⟩   outputs
} Event
```

Chapter 6

Proof of Concept

In this section, I present two experiments I did for the proof of concept of the communication analysis though execution traces.

These experiments aimed to test the model for communication analysis and the identification algorithms. By these experiment, it should be able to know if the captured dual_traces contain sufficient information of the communication model in Chapter3. They also verify the design of the some algorithms, for their correctness.

User case study is not included in this thesis and can be the future work. The feature prototype implementation is not evaluated and can be part of the user case study. But I used the implemented feature on Atlantis to conduct the experiments.

I first present the design of the experiments and their result. And then, I discuss the result of the experiments.

6.1 Experiments

In this section, I describe the design of the experiments. Two experiments are conducted in this research. All the programs in these two experiments were written in C++ and the source code can be found in SectionE. Our search partner DRDC executed the programs in their environment and provided the captured traces, the used .dll files along with the source code of the programs for the experiments.

Results are provided for each experiment. Both of the conducted experiments are about named pipe communication method. The following two subsections provides the details of the experiments and their result.

6.1.1 Experiment 1

In the first experiment, two programs communicated with each other through a synchronous Named pipe channel. One of the programs acted as the Named pipe server while the other as the client. Figure6.1 is the sequence diagram of the interaction between the server and client. Traces were captured while these two program were running and interacting. The two captured traces are analysis as dual_trace $exp1$ in this experiment. I used the implemented features in Atlantis to analyse this dual_trace. I ran the "Stream identification" and "Communication identification" operations for this dual_trace. The identified streams, communication and the processing time are listed in Figure6.2.

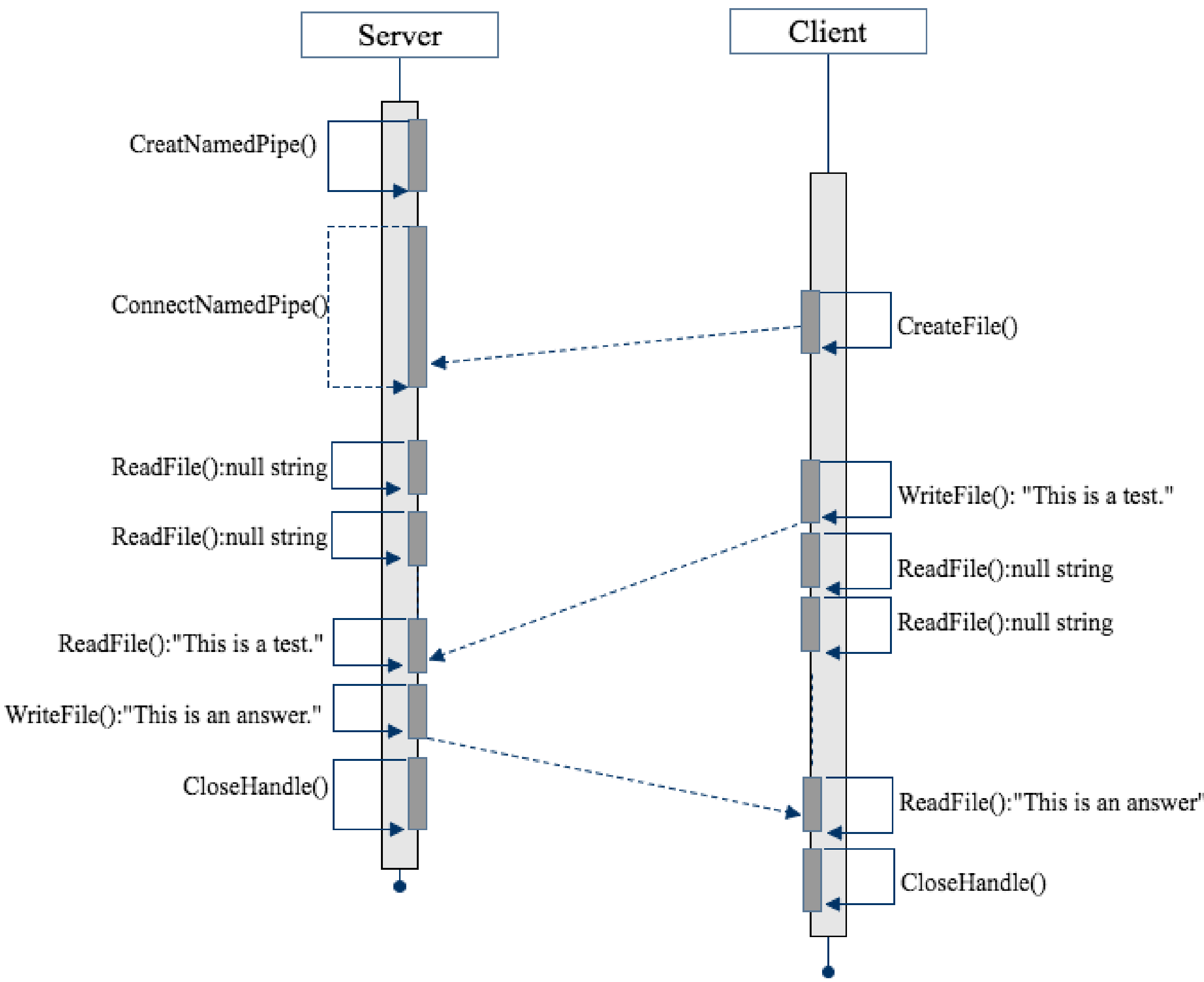

Figure 6.1: Sequence Diagram of Experiment 1

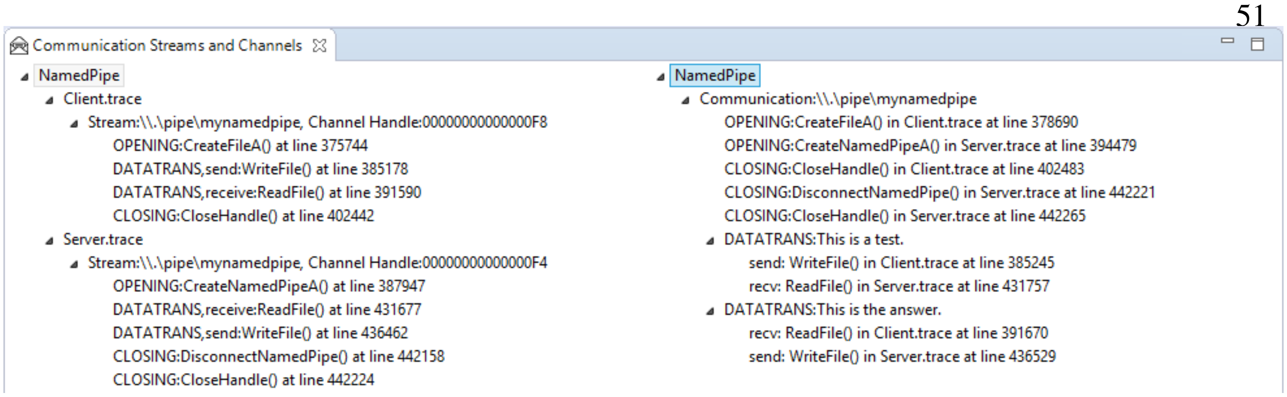

Figure 6.2: Identification result of $exp1$

6.1.2 Experiment 2

In the second experiment, one program was running as the Named pipe server. In this server program, four named pipes were created and can be connected by up to four client at a time. Two other programs as the Named pipe clients connected to this server. Those two clients (client 1 and client 2) used the identical program but run in sequence. Figure6.3 is the sequence diagram of the interaction among the server and clients. The function calls' sequence is only a possible combination from analyzing the source code. The real happening sequence can be vary from program run to program run. Traces were captured at the time when these five programs were running and interacting. One trace for each program. I only analyzed three traces which are considered as two dual_traces, $exp2.1$ and $exp2.2$. $exp2.1$ consist of traces of server and client 1 and $exp2.2$ consist of traces of server and client 2. I also ran the "Stream identification" and "Communication identification" operations for these two dual_trace. The identified streams, communication and the processing time are listed in Figure6.4 and Figure6.5.

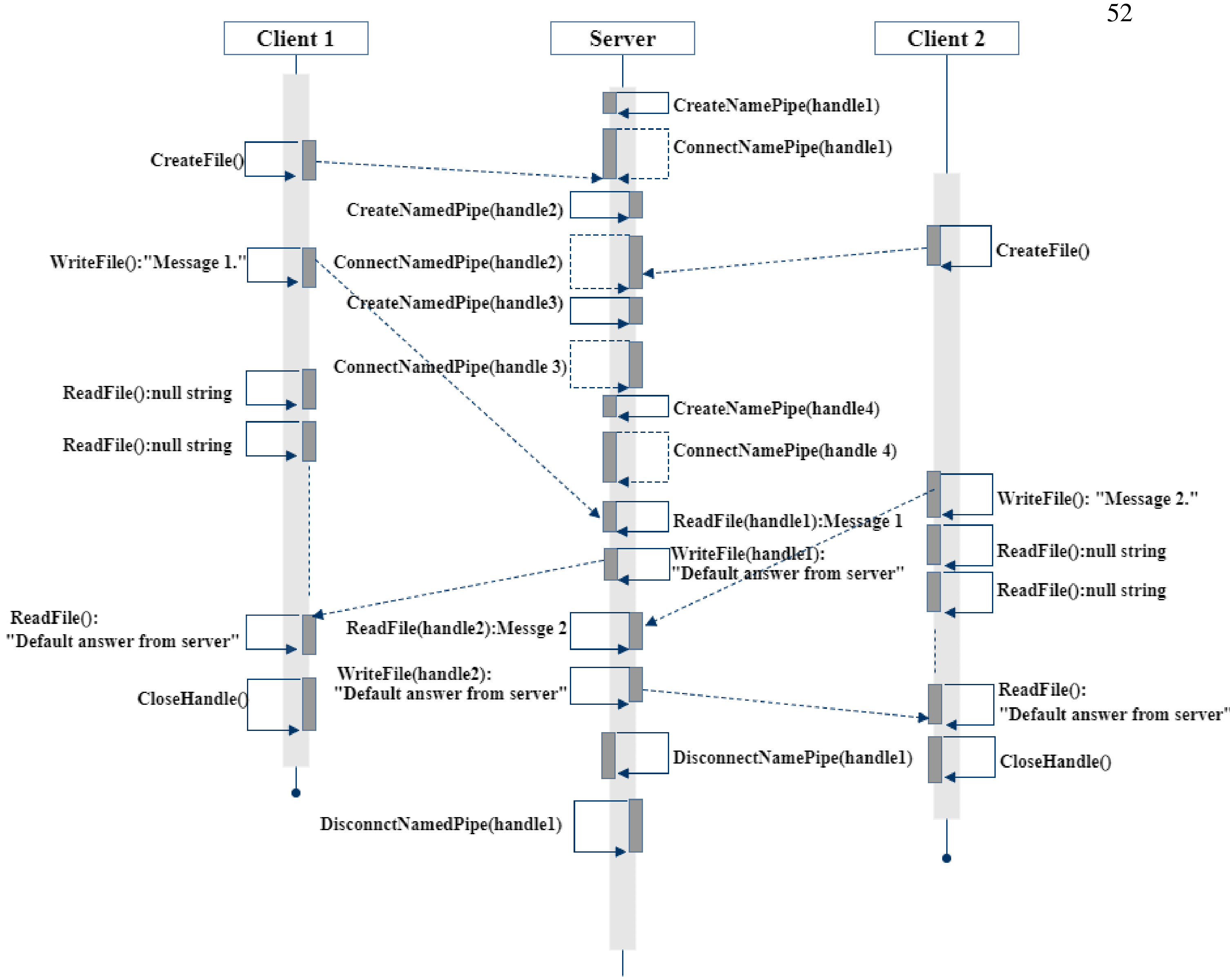

Figure 6.3: Sequence Diagram of Experiment 2

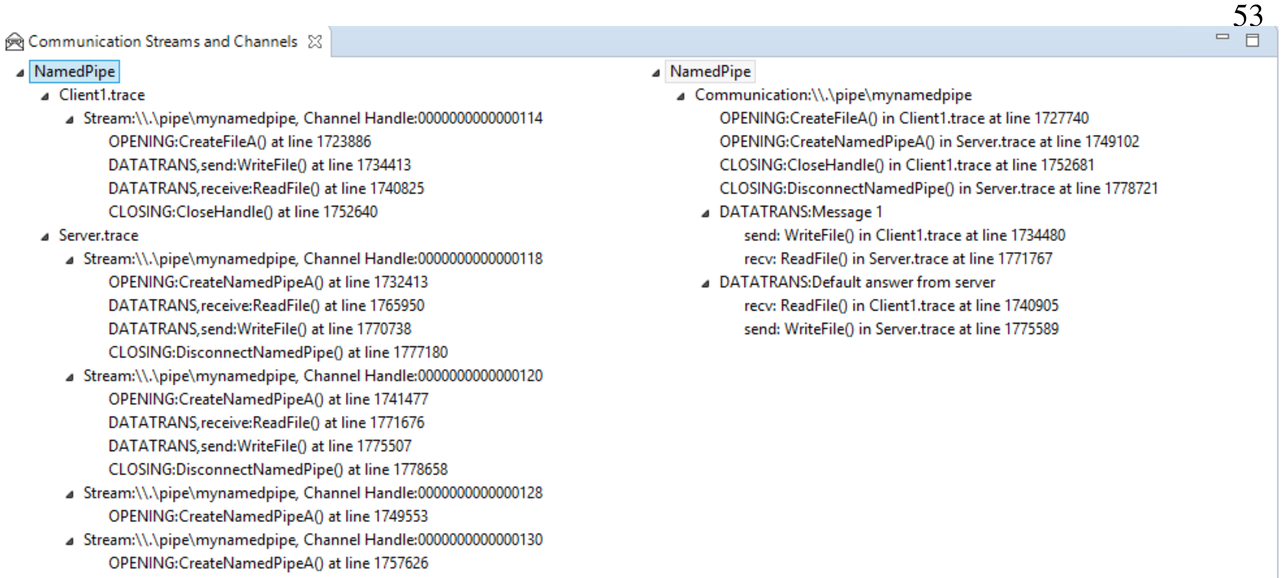

Figure 6.4: Identification result of $exp2.1$

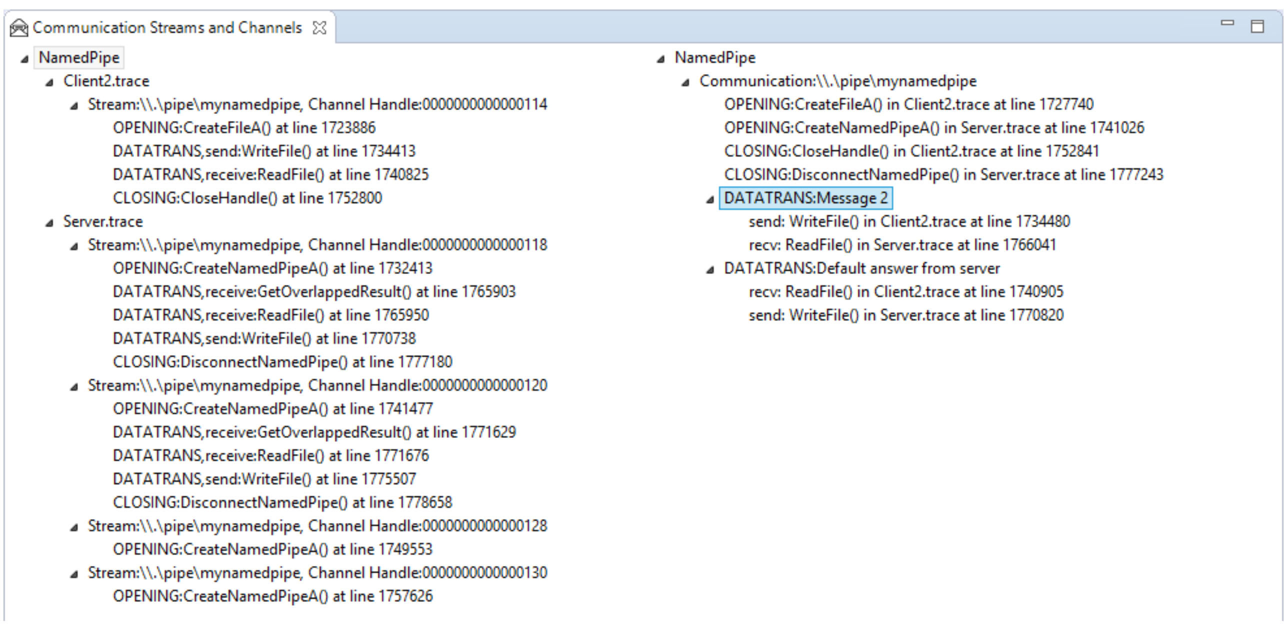

Figure 6.5: Identification result of $exp2.1$

6.2 Discussion

In the result of $exp1$, there are one stream identified in client trace and one in server trace, and these two streams are matched into a communication of this dual_trace. This identification result represents the actual communication happen between the named pipe server and client. In the result of $exp2.1$ and $exp2.2$, there are one stream identified in client traces and four in server trace respectively for each dual_trace. The streams are further matched and verified and eventually one communication is identified for each dual_trace. The result aligns to the sequence diagram in Figure6.3.

Chapter 7

Conclusions and Future Work

In this thesis, I present the designed communication identification models. This model consist of three sub models, communication definition model, dual_trace model and identification matching model for matching the elements in the dual_trace to the elements in the communication definition. This model provide the guideline for communication analysis for software security engineers and researchers in assembly execution trace level. By understanding this model, it should be possible for them to conduct their own communication analysis, identifying the concerned communication methods from the captured execution traces of interacting programs.

I also developed the essential algorithms for the communication identification. The high level algorithm is generalizable for all communication methods' identification while the stream identification and matching algorithm are distinct for each communication method according to their channel open and data transfer mechanisms. However, the developed algorithms provides clear and referable examples to develop your own algorithm for communication methods which are not discussed in this thesis.

On top of the existing execution trace analysis environment Atlantis, I implemented the communication identification features. The design provides the users a way to extend their concerned communication methods through the configuration file. The extended user interface allows the users to conduct the communication and stream identification from the dual_traces and navigate back from the identified result to the views of the trace in Atlantis. This feature prototype is a novel feature for conducting multiple trace analysis for reverse engineering at the time when this thesis was written.

The experiments conducted in this work preliminary proves the usability of the model and the algorithms. It also demonstrate the limitation for eliminating the false negative error of the communication identification. Other information is needed to assist the identification in order to improve its accuracy.

This thesis illustrates the novel idea and approach for dynamic program analysis which considerate the interaction of two programs. This idea is valuable due to the fact that programs or malware in the real world work collaboratively. The analysis of the communication and interaction of the programs provide more reliable information for vulnerability detection and program analysis.

Future work can be divided in two directions. One is extending the model to be more generalize for all kinds of interaction but not only the message transferring communications while the other is conducting user studies of the model and feature design to get a more concrete result of their usefulness.

Bibliography

[1] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004.

[2] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163. ACM, 2006.

[3] Derek Bruening. Qz: Dynamorio: Dynamic instrumentation tool platform.

[4] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. Sworddta: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences*, 21(1):10–20, 2016.

[5] B. Cleary, P. Gorman, E. Verbeek, M. A. Storey, M. Salois, and F. Painchaud. Reconstructing program memory state from multi-gigabyte instruction traces to support interactive analysis. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 42–51, October 2013.

[6] Gerald Combs. Wireshark Go Deep.

[7] Mark Dowd, John McDonald, and Justin Schuh. *Art of Software Security Assessment, The: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional., 1st edition, November 2006.

[8] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[9] José M Garrido. Inter-process communication. *Performance Modeling of Operating Systems Using Object-Oriented Simulation: A Practical Introduction*, pages 169–189, 2000.

[10] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[11] Huihui Nora Huang, Eric Verbeek, Daniel German, Margaret-Anne Storey, and Martin Salois. Atlantis: Improving the analysis and visualization of large assembly execution traces. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 623–627. IEEE, 2017.

[12] Intel. Pin - A Dynamic Binary Instrumentation Tool | Intel Software.

[13] School of Computing) Advisor (Prof. B. Kang) KAIST CysecLab (Graduate School of Information Security. c0demap/codemap: Codemap.

[14] Mujtaba Khambatti-Mujtaba. Named pipes, sockets and other ipc.

[15] Hongzhe Li, Taebeom Kim, Munkhbayar Bat-Erdene, and Heejo Lee. Software vulnerability detection using backward trace analysis and symbolic execution. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 446–454. IEEE, 2013.

[16] MultiMedia LLC. Named pipes (windows), 2017.

[17] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E*, 68:046116, Oct 2003.

[18] Arohi Redkar, Ken Rabold, Richard Costall, Scot Boyd, and Carlos Walzer. *Pro MSMQ: Microsoft Message Queue Programming*. Apress, 2004.

[19] Andreas Sailer, Michael Deubzer, Gerald Lüttgen, and Jürgen Mottok. Coretana: A trace analyzer for reverse engineering real-time software. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 657–660. IEEE, 2016.

[20] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

[21] tcpdump. Tcpdump/Libpcap public repository.

[22] Jonas Trümper, Stefan Voigt, and Jürgen Döllner. Maintenance of embedded systems: Supporting program comprehension using dynamic analysis. In *Software Engineering for Embedded Systems (SEES), 2012 2nd International Workshop on*, pages 58–64. IEEE, 2012.

[23] Chao Wang and Malay Ganai. Predicting Concurrency Failures in the Generalized Execution Traces of x86 Executables. In *Runtime Verification*, pages 4–18. Springer, Berlin, Heidelberg, September 2011. DOI: 10.1007/978-3-642-29860-8_2.

[24] Shameng Wen, Qingkun Meng, Chao Feng, and Chaojing Tang. A model-guided symbolic execution approach for network protocol implementations and vulnerability detection. *PloS one*, 12(11):e0188229, 2017.

[25] Oleh Yuschuk. Ollydbg, 2007.

[26] Dazhi Zhang, Donggang Liu, Yu Lei, David Kung, Christoph Csallner, and Wenhua Wang. Detecting vulnerabilities in c programs using trace-based testing. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 241–250. IEEE, 2010.

Appendix A

Terminology

1. **Endpoint:**An instance in a program at which a stream of data are sent or received (or both). Such as a socket handle of TCP or a file handle of the named pipe.
2. **Channel:**A conduit connected two endpoints through which data can be sent and received.
3. **Channel open event:**Operation to create and connect an endpoint to a specific channel.
4. **Channel close event:**Operation to disconnect and delete the endpoint from the channel.
5. **Send event:**Operation to send a trunk of data from one endpoint to the other through the channel.
6. **Receive event:**Operation to receive a trunk of data at one endpoint from the other through the channel.
7. **Channel open stream:**A set of all channel open events regarding to a specific endpoint.
8. **Channel close stream:**A set of all channel close events regarding to a specific endpoint.
9. **Send stream:**A set of all send events regarding to a specific endpoint.
10. **Receive stream:**A set of all receive events regarding to a specific endpoint.
11. **Stream:**A stream consist of a channel open, a channel close, a send and a receive streams of an endpoint.

Appendix B

Microsoft x64 Calling Convention for C/C++

1. RCX, RDX, R8, R9 are used for integer and pointer arguments in that order left to right.
2. XMM0, 1, 2, and 3 are used for floating point arguments.
3. Additional arguments are pushed on the stack left to right. ...
4. Parameters less than 64 bits long are not zero extended; the high bits contain garbage.
5. Integer return values (similar to x86) are returned in RAX if 64 bits or less.
6. Floating point return values are returned in XMM0.
7. Larger return values (structs) have space allocated on the stack by the caller, and RCX then contains a pointer to the return space when the callee is called. Register usage for integer parameters is then pushed one to the right. RAX returns this address to the caller.

Appendix C

Function Set Configuration Example

Listing C.1: communicationMethods.json

```
[
  {
    "communicationMethod": "NamedPipe",
    "funcList": [
      {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": true
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": false
        },
        "functionName": "CreateNamedPipeA",
        "createHandle": true,
        "type": "open"
      },
            {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": true
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": false
        },
        "functionName": "ConnectNamedPipe",
        "createHandle": false,
        "type": "open"
      },
      {
        "retrunValReg": {
          "name": "RAX",
```

```
      "valueOrAddress": true
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": false
    },
    "functionName": "CreateFileA",
    "createHandle": true,
    "type": "open"
  },
      {
    "retrunValReg": {
      "name": "RAX",
      "valueOrAddress": value
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": value
    },
    "memoryInputReg": {
      "name": "RDX",
      "valueOrAddress": address
    },
    "memoryInputLenReg": {
      "name": "R8",
      "valueOrAddress": value
    },
    "functionName": "WriteFile",
    "createHandle": false,
    "type": "send"
  },
  {
    "retrunValReg": {
      "name": "RAX",
      "valueOrAddress": value
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": value
    },
    "memoryOutputReg": {
      "name": "RDX",
      "valueOrAddress": address
    },
    "memoryOutputBufLenReg": {
      "name": "R8",
      "valueOrAddress": value
    },
    "functionName": "ReadFile",
    "createHandle": false,
    "type": "recv",
    "outputDataAddressIndex": "NamedPipeChannelRDX"
```

```
      },
      {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": value
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": value
        },
        "memoryOutputReg": {
          "name": "RDX",
          "valueOrAddress": address
        },
        "functionName": "GetOverlappedResult",
        "createHandle": false,
        "type": "check",
        "outputDataAddressIndex": "NamedPipeChannelRDX"
      },
          {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": value
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": value
        },
        "functionName": "CloseHandle",
        "createHandle": false,
        "type": "close"
      }
          {
        "retrunValReg": {
          "name": "RAX",
          "valueOrAddress": value
        },
        "valueInputReg": {
          "name": "RCX",
          "valueOrAddress": value
        },
        "functionName": "DisconnectNamedPipe",
        "createHandle": false,
        "type": "close"
      }
    ]
  }
]
```

Appendix D

Code of the Parallel Editors

Two essential pieces of code are listed for the parallel editor. One is for splitting the editor area for two editors while the other is to get the active parallel editors later on for dual_trace analysis.

D.1 The Editor Area Split Handler

Listing D.1: code in OpenDualEditorsHandler.java

```
public class OpenDualEditorsHandler extends AbstractHandler {
      EModelService ms;
      EPartService ps;
      WorkbenchPage page;


   public Object execute(ExecutionEvent event) throws ExecutionException {
            IEditorPart editorPart = HandlerUtil.getActiveEditor(event);
            if (editorPart == null) {
                  Throwable throwable = new Throwable("No active editor");
                  BigFileApplication.showErrorDialog("No active editor", "Please open one file
                      ↪ first", throwable);
                  return null;
            }

            MPart container = (MPart) editorPart.getSite().getService(MPart.class);
            MElementContainer m = container.getParent();
            if (m instanceof PartSashContainerImpl) {
                  Throwable throwable = new Throwable("The active file is already opened in one
                      ↪ of the parallel editors");
                  BigFileApplication.showErrorDialog("TThe active file is already opened in one
                      ↪ of the parallel editors",
                              "The active file is already opened in one of the parallel editors",
                                  ↪  throwable);
                  return null;
```

```
        }
        IFile file = getPathOfSelectedFile(event);

        IEditorDescriptor desc = PlatformUI.getWorkbench().getEditorRegistry().
            ↪ getDefaultEditor(file.getName());
        try {
            IFileUtils fileUtil = RegistryUtils.getFileUtils();
            File f = BfvFileUtils.convertFileIFile(file);
            f = fileUtil.convertFileToBlankFile(f);
            IFile convertedFile = ResourcesPlugin.getWorkspace().getRoot().
                ↪ getFileForLocation(Path.fromOSString(f.getAbsolutePath()));
            convertedFile.getProject().refreshLocal(IResource.DEPTH_INFINITE, null);
            if (!convertedFile.exists()) {
                createEmptyFile(convertedFile);
            }

            IEditorPart containerEditor = HandlerUtil.getActiveEditorChecked(event);
            IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindowChecked(event);
            ms = window.getService(EModelService.class);
            ps = window.getService(EPartService.class);
            page = (WorkbenchPage) window.getActivePage();
            IEditorPart editorToInsert = page.openEditor(new FileEditorInput(convertedFile)
                ↪ , desc.getId());
            splitEditor(0.5f, 3, editorToInsert, containerEditor, new FileEditorInput(
                ↪ convertedFile));
            window.getShell().layout(true, true);


        } catch (CoreException e) {
            e.printStackTrace();
        }

        return null;
    }

private void createEmptyFile(IFile file) {
        byte[] emptyBytes = "".getBytes();
        InputStream source = new ByteArrayInputStream(emptyBytes);
        try {
            createParentFolders(file);
            if(!file.exists()){
                file.create(source, false, null);
            }
        } catch (CoreException e) {
            e.printStackTrace();
        }finally{
            try {
                source.close();
            } catch (IOException e) {
                // Don't care
            }
        }
```

```
}

private void splitEditor(float ratio, int where, IEditorPart editorToInsert, IEditorPart
    ↪ containerEditor,
        FileEditorInput newEditorInput) {
    MPart container = (MPart) containerEditor.getSite().getService(MPart.class);
    if (container == null) {
        return;
    }

    MPart toInsert = (MPart) editorToInsert.getSite().getService(MPart.class);
    if (toInsert == null) {
        return;
    }

    MPartStack stackContainer = getStackFor(container);
    MElementContainer<MUIElement> parent = container.getParent();
    int index = parent.getChildren().indexOf(container);
    MStackElement stackSelElement = stackContainer.getChildren().get(index);

    MPartSashContainer psc = ms.createModelElement(MPartSashContainer.class);
    psc.setHorizontal(true);
    psc.getChildren().add((MPartSashContainerElement) stackSelElement);
    psc.getChildren().add(toInsert);
    psc.setSelectedElement((MPartSashContainerElement) stackSelElement);

    MCompositePart compPart = ms.createModelElement(MCompositePart.class);
    compPart.getTags().add(EPartService.REMOVE_ON_HIDE_TAG);
    compPart.setCloseable(true);
    compPart.getChildren().add(psc);
    compPart.setSelectedElement(psc);
    compPart.setLabel("dual-trace:" + containerEditor.getTitle() + " and " +
        ↪ editorToInsert.getTitle());

    parent.getChildren().add(index, compPart);
    ps.activate(compPart);

}

private MPartStack getStackFor(MPart part) {
    MUIElement presentationElement = part.getCurSharedRef() == null ? part : part.
        ↪ getCurSharedRef();
    MUIElement parent = presentationElement.getParent();
    while (parent != null && !(parent instanceof MPartStack))
        parent = parent.getParent();

    return (MPartStack) parent;
}


private IFile getPathOfSelectedFile(ExecutionEvent event) {
    IWorkbenchWindow window = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
```

```
            if (window != null) {
                  window = HandlerUtil.getActiveWorkbenchWindow(event);
                  IStructuredSelection selection = (IStructuredSelection) window.
                      ↪ getSelectionService().getSelection();
                  Object firstElement = selection.getFirstElement();
                  if (firstElement instanceof IFile) {
                        return (IFile) firstElement;
                  }
                  if (firstElement instanceof IFolder) {
                        IFolder folder = (IFolder) firstElement;
                        AtlantisBinaryFormat binaryFormat = new AtlantisBinaryFormat(
                                    folder.getRawLocation().makeAbsolute().toFile());
                        // arbitrary, just any file in the binary set is needed
                        return AtlantisFileUtils.convertFileIFile(binaryFormat.getExecVtableFile
                            ↪ ());
                  }
            }
            return null;
      }
}
```

D.2 Get the Active Parallel Editors

Listing D.2: code for getting parallel editors

```
IEditorPart editorPart = PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().
    ↪ getActiveEditor();
            MPart container = (MPart) editorPart.getSite().getService(MPart.class);
            MElementContainer m = container.getParent();
            if (!(m instanceof PartSashContainerImpl)) {
                  Throwable throwable = new Throwable("This is not a dual-trace");
                  BigFileApplication.showErrorDialog("This is not a dual-trace!", "Open a dual-
                      ↪ trace First", throwable);
                  return;
            }

            MPart editorPart1 = (MPart) m.getChildren().get(0);
            MPart editorPart2 = (MPart) m.getChildren().get(1);
```

Appendix E

Code of the Programs in the Experiments

E.1 Experiment 1

The two interacting programs were Named pipe server and client. The first piece of code listed below is the code for the server's program while the second piece is for the client program.

Listing E.1: NamedPipeServer.cpp

```
// Example code from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365588(v=vs.85).
    ↪ aspx

#include <Windows.h>
#include <stdio.h>
#include <strsafe.h>

#define BUFSIZE 512

DWORD WINAPI InstanceThread(LPVOID);
VOID GetAnswerToRequest(char *, char *, LPDWORD);

int main(VOID) {
      BOOL fConnected = FALSE;
      DWORD dwThreadId = 0;
      HANDLE hPipe = INVALID_HANDLE_VALUE, hThread = NULL;
      char *lpszPipename = "\\\\.\\pipe\\mynamedpipe";

      // The main loop creates an instance of the named pipe and
      // then waits for a client to connect to it. When the client
      // connects, a thread is created to handle communications
      // with that client, and this loop is free to wait for the
      // next client connect request. It is an infinite loop.
      for (;;) {
            hPipe = CreateNamedPipe(
                  lpszPipename,    // pipe name
                  PIPE_ACCESS_DUPLEX,   // read/write access
```

```
            PIPE_TYPE_MESSAGE |   // message type pipe
            PIPE_READMODE_MESSAGE |  // message-read mode
            PIPE_WAIT,     // blocking mode
            PIPE_UNLIMITED_INSTANCES, // max. instances
            BUFSIZE,     // output buffer size
            BUFSIZE,     // input buffer size
            0,       // client time-out
            NULL);      // default security attribute

        if (hPipe == INVALID_HANDLE_VALUE) {
            return -1;
        }

        // Wait for the client to connect; if it succeeds,
        // the function returns a nonzero value. If the function
        // returns zero, GetLastError returns ERROR_PIPE_CONNECTED.
        fConnected = ConnectNamedPipe(hPipe, NULL) ? TRUE : (GetLastError() ==
            ↪ ERROR_PIPE_CONNECTED);

        if (fConnected) {
            // Create a thread for this client
            hThread = CreateThread(
                NULL,    // no security attribute
                0,     // default stack size
                InstanceThread,  // thread proc
                (LPVOID)hPipe,  // thread parameter
                0,     // not suspended
                &dwThreadId);  // returns thread ID

            if (hThread == NULL) {
                return -1;
            }
            else CloseHandle(hThread);
        }
        else
            // The client could not connect, so close the pipe.
            CloseHandle(hPipe);
    }
    return 0;
}

// This routine is a thread processing function to read from and reply to a client
// via the open pipe connection passed from the main loop. Note this allows
// the main loop to continue executing, potentially creating more theads of
// this procedure to run concurrently, depending on the number of incoming
// client connections.
DWORD WINAPI InstanceThread(LPVOID lpvParam) {
    HANDLE hHeap = GetProcessHeap();
    char *pchRequest = (char *)HeapAlloc(hHeap, 0, BUFSIZE);
    char *pchReply = (char *)HeapAlloc(hHeap, 0, BUFSIZE);

    DWORD cbBytesRead = 0, cbReplyBytes = 0, cbWritten = 0;
```

```
BOOL fSuccess = FALSE;
HANDLE hPipe = NULL;

// Do some extra error checking since the app will keep running even if this
// thread fails.
if (lpvParam == NULL) {
      if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
      if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
      return (DWORD)-1;
}

if (pchRequest == NULL) {
      if (pchReply != NULL) HeapFree(hHeap, 0, pchReply);
      return (DWORD)-1;
}

if (pchReply == NULL) {
      if (pchRequest != NULL) HeapFree(hHeap, 0, pchRequest);
      return (DWORD)-1;
}

// The thread's parameter is a handle to a pipe object instance.
hPipe = (HANDLE)lpvParam;

// Loop until done reading
while (1) {
      // Read client requests from the pipe. This simplistic code only allows messages
      // up to BUFSIZE characters in length.
      fSuccess = ReadFile(
            hPipe,   // handle to pipe
            pchRequest,  // buffer to receive data
            BUFSIZE,  // size of buffer
            &cbBytesRead, // number of bytes read
            NULL);

      if (!fSuccess || cbBytesRead == 0) {
            break;
      }

      // Process the incoming message.
      GetAnswerToRequest(pchRequest, pchReply, &cbReplyBytes);

      // Write the reply to the pipe.
      fSuccess = WriteFile(
            hPipe,   // handle to pipe
            pchReply,  // buffer to write from
            cbReplyBytes, // number of bytes to write
            &cbWritten,  // number of bytes written
            NULL);   // not overlapped I/O

      if (!fSuccess || cbReplyBytes != cbWritten) {
            break;
```

```
            }
        }

        // Flush the pipe to allow the client to read the pipe's contents
        // before disconnecting. Then disconnect the pipe, and close the
        // handle to this pipe instance.
        FlushFileBuffers(hPipe);
        DisconnectNamedPipe(hPipe);
        CloseHandle(hPipe);

        HeapFree(hHeap, 0, pchRequest);
        HeapFree(hHeap, 0, pchReply);
        return 1;
}

// This routine is a simple function to print the client request to the console
// and populate the reply buffer with a default data string. This is where you
// would put the actual client request processing code that runs in the context
// of an instance thread. Keep in mind the main thread will continue to wait for
// and receive other client connections while the instance thread is working.
VOID GetAnswerToRequest(char *pchRequest, char *pchReply, LPDWORD pchBytes) {
        printf("Client␣Request␣String:\"%s\"\n", pchRequest);

        // Check the outgoing message to make sure it's not too long for the buffer.
        if (FAILED(StringCchCopy(pchReply, BUFSIZE, "This␣is␣the␣answer."))) {
                *pchBytes = 0;
                pchReply[0] = 0;
                return;
        }
        *pchBytes = lstrlen(pchReply) + 1;
}
```

Listing E.2: NamedPipeClient.cpp

```
// Example code from: https://msdn.microsoft.com/en-us/library/windows/desktop/aa365592(v=vs.85).
    ↪ aspx

#include <Windows.h>
#include <stdio.h>
#include <conio.h>

#define BUFSIZE 512

int main(int argc, char *argv[]) {
        HANDLE hPipe;
        char* lpvMessage = "This␣is␣a␣test.";
        char chBuf[BUFSIZE];
        BOOL fSuccess = FALSE;
        DWORD cbRead, cbToWrite, cbWritten, dwMode;
        char* lpszPipename = "\\\\.\\pipe\\mynamedpipe";

        if (argc > 1)
```

```
        lpvMessage = argv[1];

    // Try to open a named pipe; wait for it, if necessary.
    while (1) {
        hPipe = CreateFile(
            lpszPipename, // pipe name
            GENERIC_READ | // read and write access
            GENERIC_WRITE,
            0,    // no sharing
            NULL,   // default security attributes
            OPEN_EXISTING, // opens existing pipe
            0,    // default attributes
            NULL);   // no template file

        // Break if the pipe handle is valid.
        if (hPipe != INVALID_HANDLE_VALUE)
            break;

        // Exit if an error other than ERROR_PIPE_BUSY occurs.
        if (GetLastError() != ERROR_PIPE_BUSY) {
            return -1;
        }

        // All pipe instances are busy, so wait for 20 seconds.
        if (!WaitNamedPipe(lpszPipename, 20000)) {
            return -1;
        }
    }

    // The pipe connected; change to message-read mode.
    dwMode = PIPE_READMODE_MESSAGE;
    fSuccess = SetNamedPipeHandleState(
        hPipe,  // pipe handle
        &dwMode, // new pipe mode
        NULL,  // don't set maximum bytes
        NULL);  // don't set maximum time

    if (!fSuccess) {
        return -1;
    }

    // Send a message to the pipe server.
    cbToWrite = (lstrlen(lpvMessage) + 1);

    fSuccess = WriteFile(
        hPipe,   // pipe handle
        lpvMessage,  // message
        cbToWrite,  // message length
        &cbWritten,  // bytes written
        NULL);   // not overlapped

    if (!fSuccess) {
```

```
            return -1;
      }

      do {
            // Read from the pipe.
            fSuccess = ReadFile(
                  hPipe,  // pipe handle
                  chBuf,  // buffer to receive reply
                  BUFSIZE, // size of buffer
                  &cbRead, // number of bytes read
                  NULL);

            if (!fSuccess && GetLastError() != ERROR_MORE_DATA)
                  break;

      } while (!fSuccess); // repeat loop if ERROR_MORE_DATA

      if (!fSuccess) {
            return -1;
      }

      getch();
      CloseHandle(hPipe);

      return 0;
}
```

E.2 Experiment 2

In the experiment 2, two clients run the same program in sequence to connect to the server with asynchronous Named pipe channel. The first piece of code listed below is the code for the server's program while the second piece is the test.bat is the script for running the experiment. The client program's code is identical to experiment 1.

Listing E.3: NamedPipeServerOverlapped.cpp

```
#include <Windows.h>
#include <stdio.h>
#include <strsafe.h>

#define CONNECTING_STATE 0
#define READING_STATE 1
#define WRITING_STATE 2
#define INSTANCES 4
#define PIPE_TIMEOUT 5000
#define BUFSIZE 4096

unsigned int ReplyCount = 0;
```

```
typedef struct {
      OVERLAPPED oOverlap;
      HANDLE hPipeInst;
      char chRequest[BUFSIZE];
      DWORD cbRead;
      char chReply[BUFSIZE];
      DWORD cbToWrite;
      DWORD dwState;
      BOOL fPendingIO;
} PIPEINST, *LPPIPEINST;

VOID DisconnectAndReconnect(DWORD);
BOOL ConnectToNewClient(HANDLE, LPOVERLAPPED);
VOID GetAnswerToRequest(LPPIPEINST);
PIPEINST Pipe[INSTANCES];
HANDLE hEvents[INSTANCES];

int main(VOID)
{
      DWORD i, dwWait, cbRet, dwErr;
      BOOL fSuccess;
      LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\mynamedpipe");
      // The initial loop creates several instances of a named pipe
      // along with an event object for each instance. An
      // overlapped ConnectNamedPipe operation is started for
      // each instance.
      for (i = 0; i < INSTANCES; i++)
      {
            // Create an event object for this instance.
            hEvents[i] = CreateEvent(
                  NULL, // default security attribute
                  TRUE, // manual-reset event
                  TRUE, // initial state = signaled
                  NULL); // unnamed event object

            if (hEvents[i] == NULL)
            {
                  return 0;
            }

            Pipe[i].oOverlap.hEvent = hEvents[i];
            Pipe[i].hPipeInst = CreateNamedPipe(
                  lpszPipename, // pipe name
                  PIPE_ACCESS_DUPLEX | // read/write access
                  FILE_FLAG_OVERLAPPED, // overlapped mode
                  PIPE_TYPE_MESSAGE | // message-type pipe
                  PIPE_READMODE_MESSAGE | // message-read mode
                  PIPE_WAIT, // blocking mode
                  INSTANCES, // number of instances
                  BUFSIZE*sizeof(TCHAR), // output buffer size
                  BUFSIZE*sizeof(TCHAR), // input buffer size
```

```
            PIPE_TIMEOUT, // client time-out
            NULL); // default security attributes

        if (Pipe[i].hPipeInst == INVALID_HANDLE_VALUE)
        {
            return 0;
        }

        // Call the subroutine to connect to the new client
        Pipe[i].fPendingIO = ConnectToNewClient(Pipe[i].hPipeInst, &Pipe[i].oOverlap);
        Pipe[i].dwState = Pipe[i].fPendingIO ? CONNECTING_STATE : READING_STATE;
    }

    while (1)
    {
        // Wait for the event object to be signaled, indicating
        // completion of an overlapped read, write, or
        // connect operation.
        dwWait = WaitForMultipleObjects(
            INSTANCES, // number of event objects
            hEvents, // array of event objects
            FALSE, // does not wait for all
            INFINITE); // waits indefinitely

        // dwWait shows which pipe completed the operation.
        i = dwWait - WAIT_OBJECT_0; // determines which pipe
        if (i < 0 || i >(INSTANCES - 1))
        {
            printf("Index␣out␣of␣range.\n");
            return 0;
        }

        // Get the result if the operation was pending.
        if (Pipe[i].fPendingIO)
        {
            fSuccess = GetOverlappedResult(
                Pipe[i].hPipeInst, // handle to pipe
                &Pipe[i].oOverlap, // OVERLAPPED structure
                &cbRet, // bytes transferred
                FALSE); // do not wait

            switch (Pipe[i].dwState)
            {
                // Pending connect operation
            case CONNECTING_STATE:
                if (!fSuccess)
                {
                    return 0;
                }
                Pipe[i].dwState = READING_STATE;
                break;
                // Pending read operation
```

```
        case READING_STATE:
              if (!fSuccess || cbRet == 0)
              {
                    DisconnectAndReconnect(i);
                    continue;
              }
              Pipe[i].cbRead = cbRet;
              Pipe[i].dwState = WRITING_STATE;
              break;
              // Pending write operation
        case WRITING_STATE:
              if (!fSuccess || cbRet != Pipe[i].cbToWrite)
              {
                    DisconnectAndReconnect(i);
                    continue;
              }
              Pipe[i].dwState = READING_STATE;
              break;
        default:
        {
              return 0;
        }
        }
  }

  // The pipe state determines which operation to do next.
  switch (Pipe[i].dwState)
  {
        // READING_STATE:
        // The pipe instance is connected to the client
        // and is ready to read a request from the client.
  case READING_STATE:
        fSuccess = ReadFile(
              Pipe[i].hPipeInst,
              Pipe[i].chRequest,
              BUFSIZE*sizeof(TCHAR),
              &Pipe[i].cbRead,
              &Pipe[i].oOverlap);

        // The read operation completed successfully.
        if (fSuccess && Pipe[i].cbRead != 0)
        {
              Pipe[i].fPendingIO = FALSE;
              Pipe[i].dwState = WRITING_STATE;
              continue;
        }

        // The read operation is still pending.
        dwErr = GetLastError();
        if (!fSuccess && (dwErr == ERROR_IO_PENDING))
        {
              Pipe[i].fPendingIO = TRUE;
```

```
                    continue;
                }

                // An error occurred; disconnect from the client.
                DisconnectAndReconnect(i);
                break;

                // WRITING_STATE:
                // The request was successfully read from the client.
                // Get the reply data and write it to the client.
            case WRITING_STATE:
                GetAnswerToRequest(&Pipe[i]);

                fSuccess = WriteFile(
                    Pipe[i].hPipeInst,
                    Pipe[i].chReply,
                    Pipe[i].cbToWrite,
                    &cbRet,
                    &Pipe[i].oOverlap);

                // The write operation completed successfully.
                if (fSuccess && cbRet == Pipe[i].cbToWrite)
                {
                    Pipe[i].fPendingIO = FALSE;
                    Pipe[i].dwState = READING_STATE;
                    continue;
                }

                // The write operation is still pending.
                dwErr = GetLastError();
                if (!fSuccess && (dwErr == ERROR_IO_PENDING))
                {
                    Pipe[i].fPendingIO = TRUE;
                    continue;
                }

                // An error occurred; disconnect from the client.
                DisconnectAndReconnect(i);
                break;

            default:
            {
                return 0;
            }
            }
        }

    return 0;
}

// DisconnectAndReconnect(DWORD)
// This function is called when an error occurs or when the client
```

```
// closes its handle to the pipe. Disconnect from this client, then
// call ConnectNamedPipe to wait for another client to connect.
VOID DisconnectAndReconnect(DWORD i)
{
      // Disconnect the pipe instance.
   DisconnectNamedPipe(Pipe[i].hPipeInst)
      // Call a subroutine to connect to the new client.
      Pipe[i].fPendingIO = ConnectToNewClient(Pipe[i].hPipeInst, &Pipe[i].oOverlap);
      Pipe[i].dwState = Pipe[i].fPendingIO ? CONNECTING_STATE : READING_STATE;
}

// ConnectToNewClient(HANDLE, LPOVERLAPPED)
// This function is called to start an overlapped connect operation.
// It returns TRUE if an operation is pending or FALSE if the
// connection has been completed.
BOOL ConnectToNewClient(HANDLE hPipe, LPOVERLAPPED lpo)
{
      BOOL fConnected, fPendingIO = FALSE;

      // Start an overlapped connection for this pipe instance.
      fConnected = ConnectNamedPipe(hPipe, lpo);
      // Overlapped ConnectNamedPipe should return zero.
      if (fConnected) {
            return 0;
      }

      // Sleep random time for overlap
      Sleep(1000 * (1 + rand() % 4));

      switch (GetLastError()) {
      // The overlapped connection is in progress.
      case ERROR_IO_PENDING:
            fPendingIO = TRUE;
            break;
      // Client is already connected, so signal an event
      case ERROR_PIPE_CONNECTED:
            if (SetEvent(lpo->hEvent))
                  break;
      // If an error occurs during the connect operation...
      default:
      {
            return 0;
      }
      }
      return fPendingIO;
}

void GetAnswerToRequest(LPPIPEINST pipe)
{
      unsigned int currentCount = ReplyCount;
      ReplyCount++;
      StringCchCopy(pipe->chReply, BUFSIZE, "Answer␣from␣server");
```

```
    pipe->cbToWrite = lstrlen(pipe->chReply) + 1;
}
```

Listing E.4: test.bat

```
@echo off
start "Server" NamedPipeServerOverlapped.exe

start "Client 1" NamedPipeClient.exe "Message 1"
start "Client 2" NamedPipeClient.exe "Message 2"
```