Communication Detection and Data Transfer Event Synchronization from Dual Trace

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

Communication Detection and Data Transfer Event Synchronization from Dual Trace

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. German. Supervisor Main, Supervisor

(Department of Same As Candidate)

Dr. M. Member One, Departmental Member

(Department of Same As Candidate)

Dr. Member Two, Departmental Member

(Department of Same As Candidate)

Dr. Outside Member, Outside Member

(Department of Not Same As Candidate)

Supervisory Committee

Dr. German. Supervisor Main, Supervisor
(Department of Same As Candidate)

Dr. M. Member One, Departmental Member
(Department of Same As Candidate)

Dr. Member Two, Departmental Member
(Department of Same As Candidate)

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:

my cat, Star Trek, and the weather, for supporting me in the low moments.

Supervisor Main, for mentoring, support, encouragement, and patience.

Grant Organization Name, for funding me with a Scholarship.

> *I believe I know the only cure, which is to make one's centre of life inside of one's self, not selfishly or excludingly, but with a kind of unassailable serenity-to decorate one's inner house so richly that one is content there, glad to welcome any one who wants to come and stay, but happy all the same in the hours when one is inevitably alone.*
>
> Edith Wharton

DEDICATION

Just hoping this is useful!

Chapter 1

Introduction

Many network application vulnerabilities occur not just in one application, but in how they interact with other systems. These kinds of vulnerabilities can be difficult to analyze. Dual-trace analysis is one approach that helps the security engineers to detect the vulnerabilities in the interactive software. A dual-trace consist of two execution traces that are generated from two interacting applications. Each of these traces contains information including CPU instructions, register and memory changes of the running application. Communication information of the interacting applications is captured as the register or memory changes on their respective traced sides.

This work is focusing on helping reverse engineers for interacting software vulnerabilities detection. We first investigated and modeled four types of commonly used channels in Windows communication foundation in order to help the reverse engineers to understand the APIs, the scenarios and the assembly trace related perspectives of these channels. Then we built a tool prototype for the communication event locating and visualization of dual-traces. Finally, we design an experiment to test our prototype and evaluate its practicality.

add an section to summarize the conclusion later

The Methodology used for this work composed of 7 major steps. To make this work executable, 1)I defined the problem by understanding the requirement from our research partner DRDC. 2) I obtained the related background knowledge by literature review. Then 3) I model the abstract communication channels. Based on these channel models,4) I develop algorithms to synchronize the communication events happen in the channel. After that, 5) I match the real channels used in Windows Communication Foundation to my channel models, verify their consistency with my models. Finally 6)I implement the synchronization algorithms for the dual-trace analysis and verify them by the dual-traces from DRDC.

1.1 Define the Problem

A dual-trace consists of two execution traces that are generated from two interacting applications. The trace analysis is based only on the assembly level execution trace which contain the instructions and memory change of a running application. Beside all the factors in single trace analysis, dual-trace analysis has to analyze the communications of the applications in the traces. A communication between two applications including the communication channel open, all data exchanging events, the communication channel close. Correspondingly, a full communication definition in the dual-trace should consist of the channel opening events in both sides, data sending and receiving events, and the the channel closing events in both sides. Each of these events consist of function call and related data from the memory record. In some cases there might be some events lacking from the trace, such as no data exchange after a channel is open, or the traces end before the channel was closed. However, the channel open is critical, without that there is no way to locate all other events in the traces. The goal communication analysis of dual-trace is to rebuild all the user concerned communication channels from the dual-trace.

1.2 Model the Communications and the Dual_Trace

There are two abstract models for communication based on the communication behavior. One is the order guaranteed communication model and the other is order in-guaranteed communication model. I define how the communication happens as well as all the data send/receive scenario in each model. Later on the real communication channels will be categorized into these two models.

1.3 Develop Communication Identification Algorithms for Dual_trace

1.4 Implement Feature Prototype on Atlantis

1.5 Evaluation of the Algorithms by Sample Dual_traces

Chapter 2

Background

This section introduces several background knowledge or information that related to this work. First I describe what is software security and how important it is as well as our previous approach to assist detection of software vulnerabilities by assembly level trace analysis. Second, I introduce the general assembly level trace as well as some tracer to generate it. Third, I discuss how software interaction affect the behavior of the software and how they related to the software vulnerabilities. Then I talk about Windows Communication Foundation in which the communication channels type used are targeted by this work. Finally, we mention some important Windows function calling conventions without which you can not picture what the function calls look like in the assembly level.

2.1 Obtain Background Knowledge

I did a some background reading in the reverse engineering filed, focusing more on the vulnerabilities detection domain to better understand the current state and needs. In addition, to locate the communication event of the dual-trace, I need to investigate the communication methods' APIs to understand their structure in the assembly level traces. I need to know how the functions for channel setup and the functions for messages sending/receiving work. The system functions I was looking for is in C++ level. I have to know the C++ function names, related parameters, return value and so on. Furthermore, to understand their structure in the assembly level trace, I have to know the calling conventions in assembly, such registers/memory for parameters or return value.

2.2 Software Security

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 vulnerabilities of the software running on it. Accordingly, the emphasize placed on computer security particularly in the field of software vulnerabilities detection increases dramatically. It's important for software developers to build secure applications. Unfortunetely, this is usually very expensive and time consuming and somehow impossible. On the other hand, finding issues in the built applications is more important and practical. Howerver this is a complex process and require deep technical understanding in the perspetive of reverse engineering.[**?**].

2.3 Software Vulnerability Detection

A common approach to detect existing vulnerabilities is fuzzing testing, which record the execution trace while supplying the program with input data up to the crash and perform the analysis of the trace to find the root cause of the crash and decide if that is a vulnerability[**?**]. Execution trace can be captured in different levels, for example object level and function level. But my research only focus on those that captured in instruction and memory reference level. There are two main reasons for analysis system-level traces. First, it is for analysis of the software provided by vendor whose source code are not available. The second one is that low level trace are more accurately reflect the instructions that are executed by multicore hardware[**?**].

2.4 Program Execution Tracing in Assembly Level

There are many tools that can trace a running program in assembly instruction level. IDA pro [**?**] 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 [**?**] to provide more sufficient features for "run-trace" visualization. PIN[**?**] 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 **??** and

2.5   Program Communications

Applications nowaday do not alway work isolately, many software appear as reticula collaborating systems connecting different modules in the network[**?**] which make the discovery of vulnerabilities even harder. The communication and interaction between modules affect the behaviour of the software. Without regarding to the synergy information , analysis of the isolated execution trace on a single computer is usually futile.

2.6   Atlantis

Applications nowaday do not alway work isolately, many software appear as reticula collaborating systems connecting different modules in the network[**?**] which make the discovery of vulnerabilities even harder. The communication and interaction between modules affect the behaviour of the software. Without regarding to the synergy information , analysis of the isolated execution trace on a single computer is usually futile.

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 Table3.2 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 always received losslessly and in order to the other endpoint. For some communication methods, a channel can be closed without waiting the completion of all data transmission. With this property, the concatenated data in the receive stream of one endpoint should be the sub string of the concatenated data in the send stream of the other endpoint. Therefore, the send and receive data verification should be in send and receive stream level by comparing the concatenated received data of one endpoint to the concatenated sent data of another.

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 should be done by matching the data packets in a receive event to 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 Table3.2 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 one-way or duplex pipe which allows two programs send and receive message through the named pipe. [2]

    The basic data transfer characteristics of Named Pipe are:

- Bytes received in order

- Bytes sent as a whole trunk can be received in segments

- No data duplication

- Only the last trunk can be lost

Based on these characteristics, the data transfer scenarios of Named pipe can be summarized in Figure3.1.
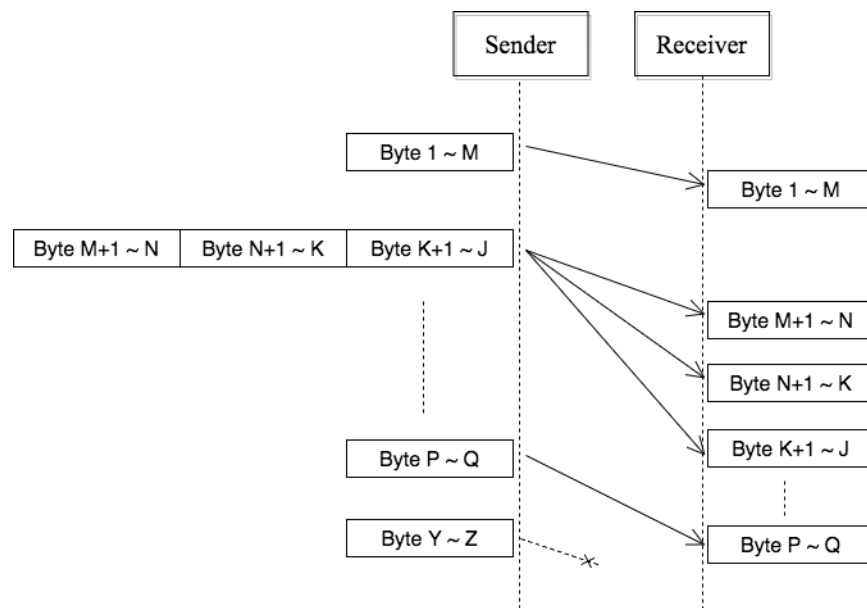
Figure 3.1: Data Transfer Scenarios for Named Pipe

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.[4] 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 summarized in Figure3.2.
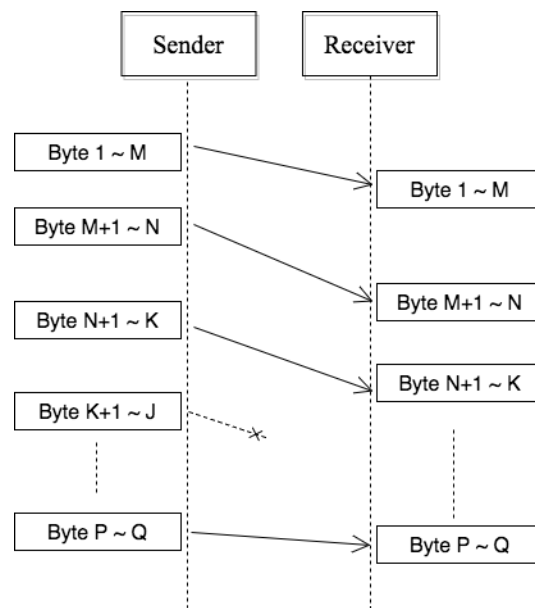
Figure 3.2: Data Transfer Scenarios for Message Queue

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 summarized in Figure3.3.
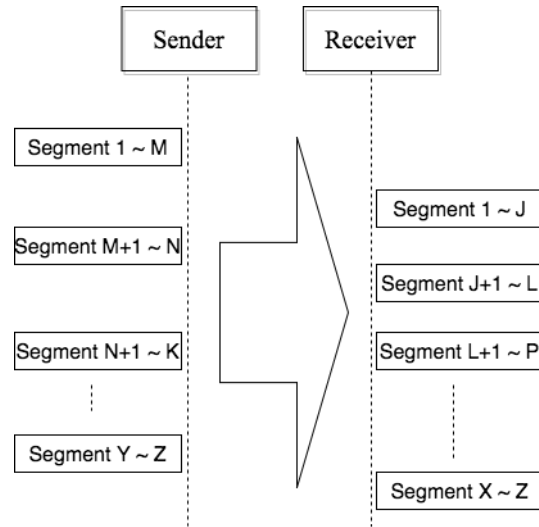
Figure 3.3: Data Transfer Scenarios for TCP

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 summarized in Figure3.4.
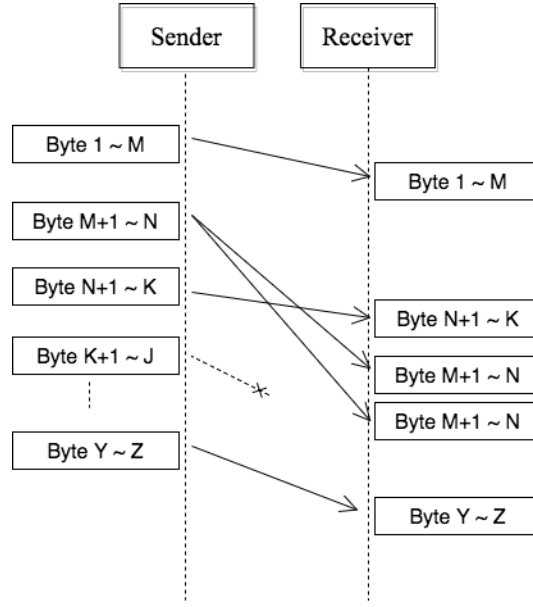
Figure 3.4: Data Transfer Scenarios for UDP

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.

A communication Co is defined by the 2-tuple $\langle ep, c \rangle$, where ep is a set $\{e_x : x = 0, 1\}$ for the two endpoints communicating with each other though the channel c which represented by a identifier. The endpoint e_x is defined by the 3-tuple $\langle h_x, ds_x, dr_x \rangle$. h_x is the handle created within a process for subsequent data transfer operations. ds_x is the sequence of packets sent in the sending operations of h_x while dr_x is the sequence of packets received in the receiving operations of h_x. e_0 is created in process p and e_1 is created in process q. Let $ds_x = (ps_{x,i} : 0 \leqslant i \leqslant I_x)$ and $dr_x = (pr_{x,j} : 0 \leqslant j \leqslant J_x)$ in which $ps_{x,i} = \langle ts_{x,i}, ss_{x,i} \rangle$ and $pr_{x,i} = \langle tr_{x,j}, sr_{x,j} \rangle$. $ts_{x,i}$ and $tr_{x,j}$ are the logical time when the packet being sent and received. $\forall ps_{x,i} \in ds_x, ts_{x,k} \leqslant tr_{x,l}$ if $k \leqslant l$;

$\forall pr_{x,i} \in dr_x$, $tr_{x,k} \leqslant tr_{x,l}$ if $k \leqslant l$; $ss_{x,i}$ and $sr_{x,j}$ are the string payloads being sent and received. The string payloads can be described as a sequence in the same order of the sequence of packets, $pls_x = (ss_{x,i} : 0 \leqslant i \leqslant I_x)$ and $plr_x = (sr_{x,j} : 0 \leqslant j \leqslant J_x)$.

There are two sets of preservation of this definition. One set is for the reliable communication while the other is for the unreliable one. There are content preservation and timing preservation in each preservation set.

Preservation for reliable communication:

- *Content Preservation:* Let S_x be the concatenation of $\forall ss_{x,i} \in pls_x$ and R_x be the concatenation of $\forall sr_{x,i} \in plr_x$. Then, R_0 is a sub string of S_1 and R_1 is a sub string of S_0.

- *Timing Preservation:* Let $S_{x,k}$ be the concatenation of $\forall ss_{x,i} \in pls_x, 0 \leqslant k \leqslant M_x$ and $R_{x,l}$ be the concatenation of $\forall sr_{x,i} \in plr_x, 0 \leqslant l \leqslant N_x$. If $S_{0,k}$ is sub string of $R_{1,l}$, then $ts_{0,k} \leqslant tr_{1,l}$. If $S_{1,k}$ is sub string of $R_{0,l}$, then $ts_{1,k} \leqslant tr_{0,l}$.

Preservation for unreliable communication:

$\forall sr_{0,j} \in plr_0, \exists ss_{1,i} \in pls_1$ and $\forall sr_{1,j} \in plr_1, \exists ss_{0,i} \in pls_0$ such that

- *Content Preservation:* $sr_{0,j} = ss_{1,i}$ and $sr_{1,j} = ss_{0,i}$

- *Timing Preservation:* $tr_{0,j} > ts_{1,i}$ and $tr_{1,j} > ts_{0,i}$

In the following two examples, h_0 and h_1 are the handles for the two endpoints of the communication. ds_0, dr_0 and ds_1, dr_1 are the sequence of packets sent and received by the endpoints. The string payloads are listed in blue and red in the figures.

Figure3.5 is an example of the reliable communication. In this example, $ss_{0,0} = $ "ab", $ss_{0,1} = $ "cde", $ss_{0,2} = $ "fgh"; $sr_{1,0} = $ "abc", $sr_{1,1} = $ "def", $ss_{1,2} = $ "gh" and on the other direction $ss_{1,0} = $ "mno", $ss_{1,1} = $ "pqr", $ss_{1,2} = $ "stu"; $sr_{0,0} = $ "mnop", $sr_{0,1} = $ "qrstu". It is clear in the example that $ss_{0,0}.ss_{0,1}.ss_{0,2} = sr_{1,0}.sr_{1,1}.ss_{1,2} = $ "abcdefgh" and $ss_{1,0}.ss_{1,1}.ss_{1,2} = sr_{0,0}.sr_{0,1} = $ "mnopqrstu". These satisfy the content preservation. The timing in this example are: $ts_{1,0} < ts_{1,1} < tr_{0,0} < ts_{1,2} < tr_{0,1}$ and $ts_{0,0} < ts_{0,1} < tr_{1,0} < ts_{1,2} < tr_{1,1} < tr_{1,2}$. The following fact of this example satisfy the timing preservation. $sr_{0,0} = $ "mnop" is the sub string of $ss_{1,0}.ss_{1,1} = $ "mnopqr", $sr_{0,0}.sr_{0,1} = $ "mnopqrstu" is the sub string of $ss_{1,0}.ss_{1,1}.ss_{1,2} = $ "mnopqr"stu, $sr_{1,0} = $ "abc" is the sub string of $ss_{0,0}.ss_{0,1} = $ "abcde", $sr_{1,0}.sr_{1,1} = $ "abcdef" and $sr_{1,0}.sr_{1,1}.sr_{1,2} = $ "abcdefgh" are the sub string of $ss_{0,0}.ss_{0,1}.ss_{0,2} = $ "abcdefg".
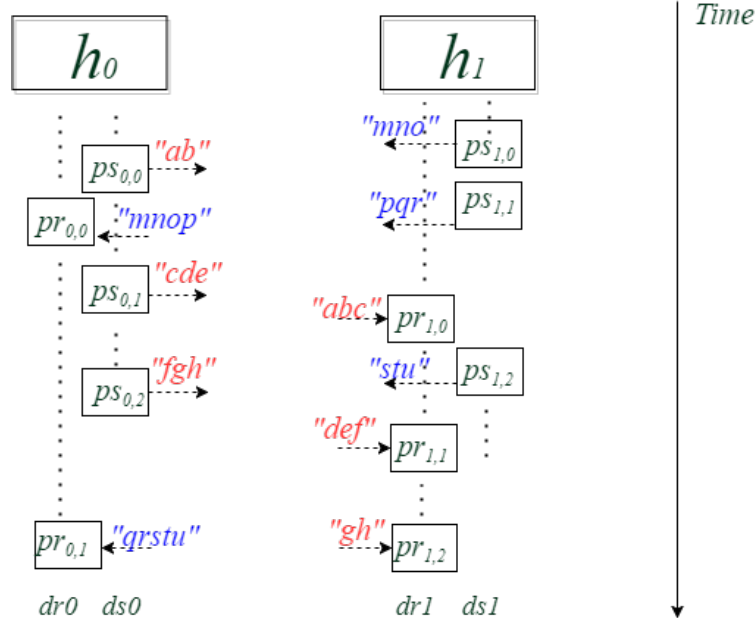
Figure 3.5: Example of Communication

Figure 3.6 is an example of the unreliable communication. In this example, $sr_{1,0} = ss_{0,1} =$ "cde", $tr_{1,0} > ts_{0,1}$; $sr_{1,1} = ss_{0,2} = $"$fi$",$tr_{1,1} > ts_{0,2}$; $sr_{0,0} = ss_{1,0} = $ "gh", $tr_{0,0} > ts_{1,0}$; $sr_{0,1} = ss_{1,1} = $ "$ijklm$", $tr_{0,1} > ts_{1,1}$; $sr_{0,2} = ss_{1,2} = $ "n", $tr_{0,2} > ts_{1,2}$. All of these satisfy the content preservation and timing preservation of the unreliable communication.
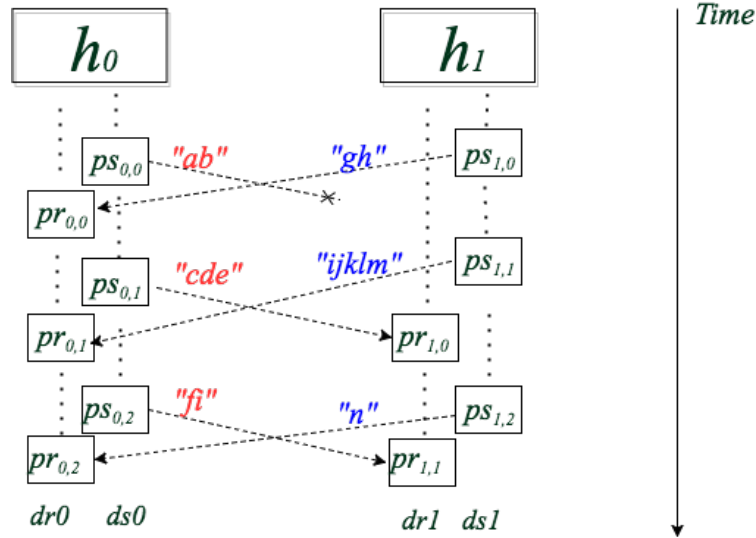
Figure 3.6: Example of Communication

3.3 Dual-Trace Model

In this section, I model the execution trace in the dual-trace. The modeling aims at identify the communications from the information summarized in the 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 $dual\_trace$ consist of two execution traces which are $\{trace_x : x = 0, 1\}$. An execution $trace$ is defined as a sequence $(line_k, 0 \leqslant k \leqslant K)$. $line_k$ in a $trace$ is a 3_tuple $\langle ins, mem, inf \rangle$ where ins is the assembly instruction, mem is memory changed by this instruction and inf is function call information. This information includes an indicator of function call and return and the function's name if applicable. A function $eventfilter\,()$ is defined to generate the event level trace $event\_trace_x$ from the original $trace_x$. So that $event\_trace_x = eventfilter\,(trace_x, funcset)$, where $funcset = \{func_l, 0 \leqslant l \leqslant L\}$ is a set of the concerned events' function information. Each concerned event's function information can be described a tuple $\langle funN, type, pars \rangle$ where $funN$ is the function name, $type$ can only be one of these four event types: channel open, channel close, data send and data receive, $pars$ is the parameter information list. The output of this function $event\_trace_x$ is a sequence of events $(event_{x,m}, 0 \leqslant m \leqslant M_x)$. Only the concerned events in the $funcset$ are filtered in this sequence, all other information in the original trace are ignored. Each event in the trace corresponds to a system function call and is defined as a 6_tuple $\langle funN, startline, endline, inputs, outputs, type \rangle$. In this tuple $funN$ is the name of the called function, $startline$ is the line number where the function was being called, $endline$ is the line number where the function returned and $type$ is the event type. The events in the $event\_trace_x$ are interleaving events among multiple handles. Function $streamfilter\,()$ is defined to generate the stream level trace $stream\_trace_x$ from the $event\_trace_x$ so that $stream\_trace_x = streamfilter\,(event\_trace_x)$. The output $stream\_trace_x$ is a set of stream $\{stream_{x,n}, 0 \leqslant n \leqslant N_x\}$ in which each stream corresponds to a handle, a channel id and consist of 4 sub streams. So $stream_{x,n}$ is a 6_tuple $\langle handle, channelid, open\_stream,$ $send\_stream, receive\_stream, close\_stream \rangle$. Each of these sub streams consist of a sequence

of *event* of a certain handle of the corresponding event types.

3.4 Element Matching Communication Model and Dual-Trace Model

The identification the communication from dual-trace can be simply abstracted as finding the elements of the communication model in the dual-trace model. The element matching can be summarized in Table.By known this matching, algorithms can be developed to identify the communications in the dual-trace model. The developed algorithm will be discussed in next chapter.

Table 3.2: Element Matching of Communication and Trace Models

Communication Model Element	Trace Model Element
c	matching $channelid$ in two $stream$ respectively of $trace_0$ and $trace_1$
ep_x	a $stream$ in $trace_x$
h_x	$handle$ of a $stream$ in $trace_x$
ds_x	a $send\_stream$ of a $stream$ in $trace_x$
dr_x	a $receive\_stream$ of a $stream$ in $trace_x$
$ps_{x,i}$	a packet send event $event_{x,m}$ in $event\_trace_x$
$ps_{x,i}$	a packet receive event $event_{x,m}$ in in $event\_trace_x$
$ss_{x,i}$	the payload can be find in $inputs$ of an send event $event_{x,m}$
$sr_{x,i}$	the payload can be find in $outputs$ of an receive event $event_{x,m}$

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 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 $dual\_trace = \{trace_x : x = 0, 1\}$ and the concerned communication method's function set $funcset = \{func_l, 0 \leqslant l \leqslant L\}$. 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: $dual\_trace$, $funcset$

    Output: $cos = \{co_y : 0 \leqslant y \leqslant Y\}$

  1 **for** $x \in (0, 1)$ **do**

  2       $event\_trace_x = eventfilter\,(trace_x, funcset)$;

  3       $stream\_trace_x = streamfilter\,(event\_trace_x)$;

  4 $cos = streammatch(stream\_trace_0, stream\_trace_1)$;

  5 **return** cos;

4.2 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 $funcset$ 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 $event\_trace$ 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.

4.2.1 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.

4.2.2 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.[3] 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

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

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.
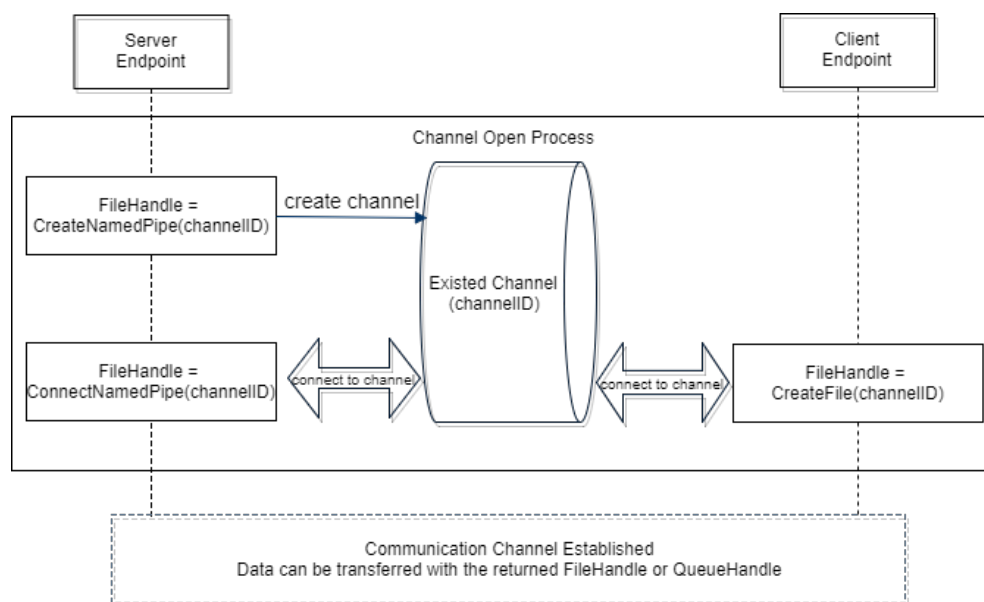
Figure 4.1: Channel Open Process for a Named Pipe

4.2.3 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.
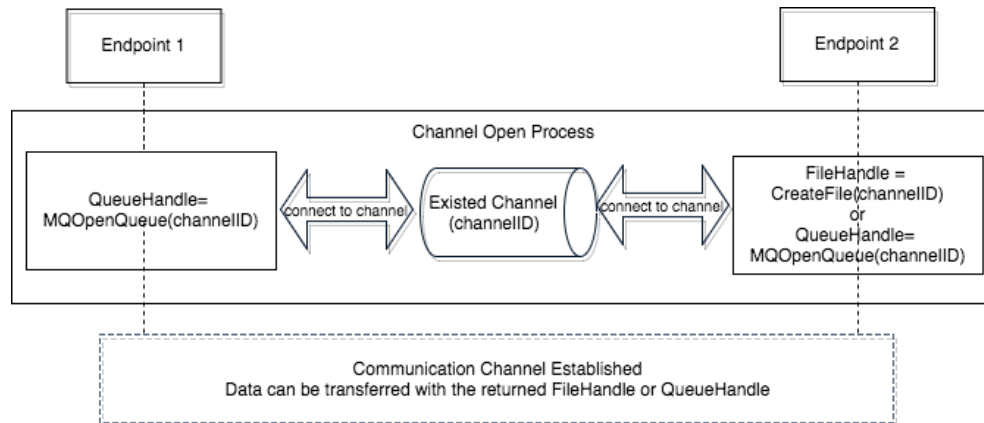
Figure 4.2: Channel Open Process for a Message Queue

4.2.4 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.
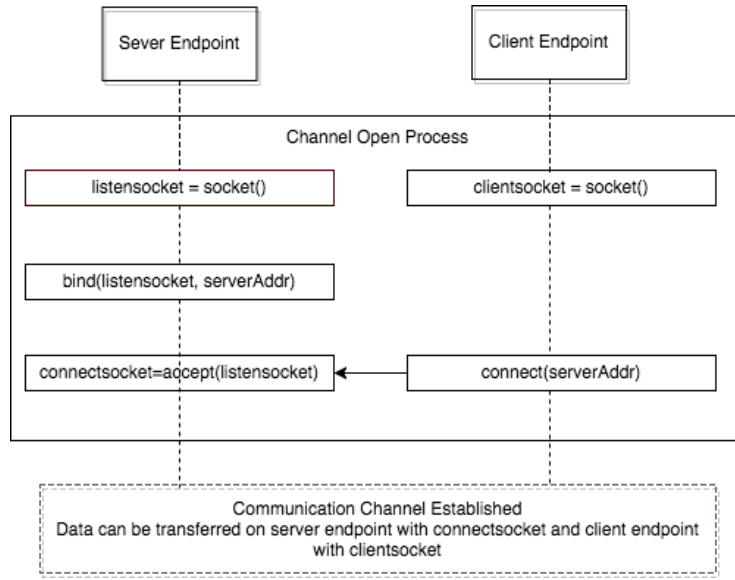
Figure 4.3: Channel Open Model for TCP and UDP

4.3 Event Locating Algorithm: $eventfilter()$

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 4.2 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 Section4.2 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: $trace, funcset$

   Output: $event\_trace$

1 $event\_trace \leftarrow List\langle Event\rangle$;

2   **while** *not at end of trace* **do**

3   **for** $f \in funcset$ **do**

4   **if** *Is function call of f* **then**

5 $event.funN = f.funN \; event.startline \leftarrow$ current Line number;

6 $event.endline \leftarrow$ find function return instruction line;

7 $event.inputs \leftarrow$ reconstruct memory of $event.startline$ from the trace and get input values of $f.pars$;

8 $event.outputs \leftarrow$ reconstruct memory of $event.endline$ from the trace and get outputs values of $f.pars$;

9 $event.type \leftarrow f.type$;

10 $event\_trace.add\,(event)$;

11 **return** $event\_trace$;

4.4   **Stream Identification Algorithm:** $stream filter\,()$

The events located in the $event\_trace$ may correspond to different $stream$, the next step in the communication identification algorithm is to identify them for each $stream$. The input of this algorithm the $event\_trace$ from the "Event Locating Algorithm". Since the input $event\_trace$ 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 $stream\_trace$. Each stream in this $stream\_trace$ consist of the sub streams. The concepts of the stream and sub streams are defined in SectionA.

Algorithm 3: Stream Indentification Algorithm

Input: $event\_trace$

Output: $stream\_trace$

1   $stream\_trace \leftarrow Map\langle String, List\langle EndPoint\rangle\rangle$;

2   **for** $event \in event\_trace$ **do**

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

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

5         $stream \leftarrow stream\_trace.get\,(handle)$;

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

7            $newHandle \leftarrow$ get the second socket handle identifier which is the return value from the function parameter list;

8            $stream\_trace.remove\,(handle)$;

9            $stream\_trace.add\,(newHandle, endpoint)$;

10   **if** *endpoint is null* **then**

11            $stream = New\ \ Stream\,()$;

12            $stream\_trace.add\,(hanele, endpoint)$;

13         $stream.openStream.add\,(event)$;

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

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

16         $stream \leftarrow stream\_trace.get\,(handle)$;

17   **if** $stream$ *is not null and* $stream.complete$ *is* $False$ **then**

18            $stream.sendStream.add\,(event)$;

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

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

21         $stream \leftarrow stream\_trace.get\,(handle)$;

22   **if** $stream$ *is not null and* $stream.complete$ *is* $False$ **then**

23            $stream.receiveStream.add\,(event)$;

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

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

26         $stream \leftarrow stream\_trace.get\,(handle)$;

27   **if** $stream$ *is not null* **then**

28            $stream.closeStream.add\,(event)$;

29            $stream \leftarrow True$;

30   **return** $stream\_trace$;

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 $stream\_trace$ from the dual-trace. The output of this algorithm is the communication list. Each communication recognized from the dual_trace contains two $stream$s. 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 $stream$s 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 $stream$s 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 $stream$s 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 $stream$s in the first level matching. If the transmitted data in the $stream$s 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.
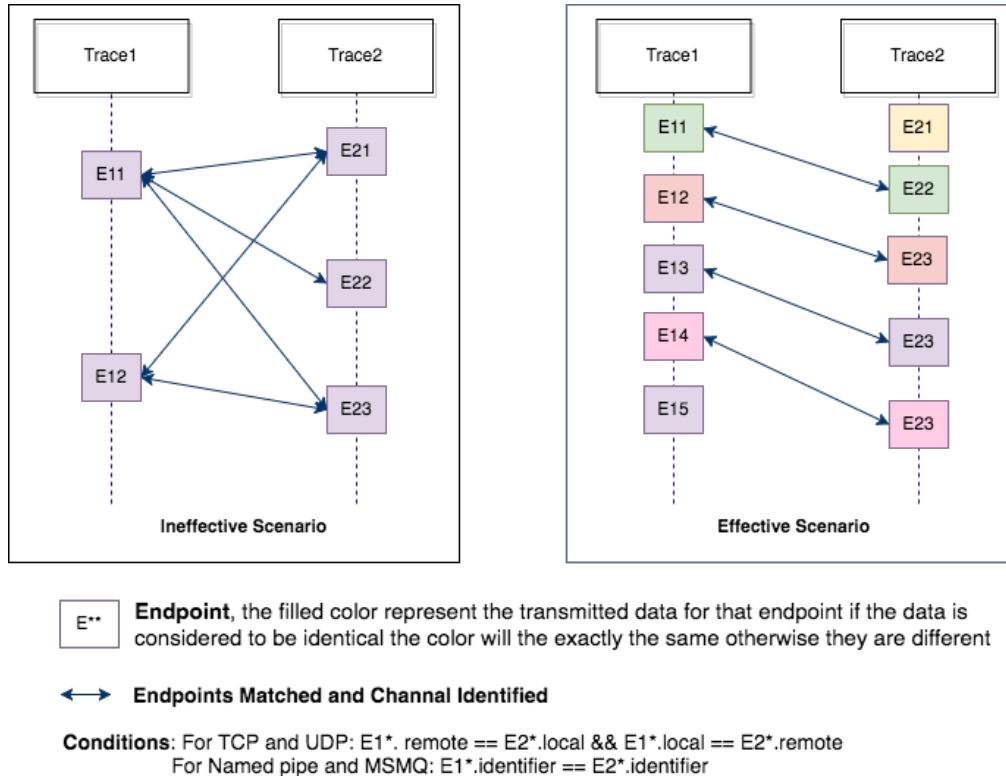
Figure 4.4: Second Level Matching Scenarios

The following subsections discuss the algorithms for these two level matching. In Section4.2, 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 $stream\_trace$s 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 $stream$s matched in the first level matching while the output a boolean to indicate if the transmitted data of this two $stream$s 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 $stream$. So in the below algorithm, when it try to get the channel open event from the $stream.openStream$ list, only one event should be found and return. The channel identifier parameters can be found in the $event.inputs$ 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: $stream\_trace_0, stream\_trace_1$

Output: $cos = \{co_y : 0 \leqslant y \leqslant Y\}$

1   $cos \leftarrow Map\langle String, List\langle Communication\rangle\rangle$;

2   **for** $stream0 \in stream\_trace_0$ **do**

3      $openEvent0 \leftarrow$ get the opening event from $stream0.openStream$;

4      $channelId0 \leftarrow$ get the channel identifier from $openEvent0.inputs$;

5   **for** $stream1 \in stream\_trace_1$ **do**

6          $openEvent1 \leftarrow$ get the opening event from $stream1.openStream$;

7          $channelId1 \leftarrow$ get the channel identifier from $openEvent1.inputs$;

8   **if** $channelId0 == channelId1$ **then**

9              $DataVerified = dataVerify(stream0, stream1, outputdata)$;

10   **if** $DataVerified == True$ **then**

11                 $communication = New\ \ Communication()$;

12                 $communication.stream0 = stream0$;

13                 $communication.stream1 = stream1$;

14                 $communication.dataMatch = outputdata$;

15                 $cos.add\,(communication)$;

16   **return** cos;

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: $stream\_trace_0, stream\_trace_1$

    Output: $cos = \{co_y : 0 \leqslant y \leqslant Y\}$

1 $cos \leftarrow Map\langle String, List\langle Communication\rangle\rangle$;

2   **for** $stream0 \in stream\_trace_0$ **do**

3 $socketEvent0 \leftarrow$ get the $socket()$ event from $stream0.openStream$;

4 $bindEvent0 \leftarrow$ get the $bind()$ event from $stream0.openStream$;

5 $connectEvent0 \leftarrow$ get the $connect()$ event from $stream0.openStream$;

6   **for** $stream1 \in stream\_trace_1$ **do**

7 $socketEvent1 \leftarrow$ get the $socket()$ event from $stream1.openStream$;

8 $bindEvent1 \leftarrow$ get the $bind()$ event from $stream1.openStream$;

9 $connectEvent1 \leftarrow$ get the $connect()$ event from $stream1.openStream$;

10   **if** $socketEvent0! = null$ AND $socketEvent1! = null$ **then**

11   **if** $bindEvent0! = null$ AND $connectEvent1 == null$ **then**

12 $localServerAddr \leftarrow$ get serverAddr from $bindEvent1.inputs$;

13   **else if** $bindEvent1 == null$ AND $connectEvent0! = null$ **then**

14 $remoteServerAddr \leftarrow$ get serverAddr from $connectEvent1.inputs$;

15   **else**

16   Break the inner For loop;

17   **if** $localServerAddr == remoteServerAddr$ **then**

18 $DataVerified = dataVerify(stream0, stream1, outputdata)$.
              $communication = New\ Communication()$;

19 $communication.stream0 = stream0$;

20 $communication.stream1 = stream1$;

21 $communication.dataMatch = outputdata$; $cos.add(communication)$;

22   **return** cos;

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 *stream*s from two traces which are being matched in the first level.

Algorithm 6: Transmitted Verification by Data Union

Input: $stream0, stream1$

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

1 **return** *Indicator of if transmitted data union are considered to be identical*

2 $send1 \leftarrow$ empty string;

3 $send2 \leftarrow$ empty string;

4 $recv1 \leftarrow$ empty string;

5 $recv2 \leftarrow$ empty string;

6 **for** $sendEvent \in stream0.sendStream$ **do**

7 $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;

8 $send0.append\,(sendmessage)$;

9 **for** $sendEvent \in stream1.sendStream$ **do**

10 $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;

11 $send1.append\,(sendmessage)$;

12 **for** $recvEvent \in stream0.receiveStream$ **do**

13 $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;

14 $recv0.append\,(sendmessage)$;

15 **for** $recvEvent \in stream1.receiveStream$ **do**

16 $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;

17 $recv1.append\,(sendmessage)$;

18 **if** $recv0$ *is substring of* $send1$ *AND* $recv1$ *is substring of* $send0$ **then**

19 **return** True;

20 **else**

21 **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 $event$s of the $sendstream$ in one $stream$ trying to find the matched receive event in the $receivestream$ 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 $stream$s 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: $stream0, stream1$

Output: matched event list of two endpoints

1 return *Indicator of if transmitted data union are considered to be identical*

2 $sendPktNum0 \leftarrow stream0.sendStream.length;$

3 $sendPktNum1 \leftarrow stream1.sendStream.length;$

4 $recvPktNum0 \leftarrow 0;$

5 $recvPktNum1 \leftarrow 0;$

6 $eventMatchs \leftarrow List\langle EventMatch \rangle;$

7 for $sendEvent \in stream0.sendStream$ **do**

8 $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs;$

9   **for** $recvEvent \in stream1.receiveStream$ **do**

10 $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs;$

11   **if** $sendmessage == recvmessage$ **then**

12 $recvPktNum0 + +;$

13 $stream1.receiveStream.remove\,(recvEvent);$

14 $eventMatch = NeweventMatch\,();$

15 $eventMatchs.add\,(eventMatch);$

16 if $(sendPktNum0 - recvPktNum0) * 100/sendPktNum0 > threshold$ **then**

17   **return** False;

18 for $sendEvent \in stream1.sendStream$ **do**

19 $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs;$

20   **for** $recvEvent \in stream0.receiveStream$ **do**

21 $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs;$

22   **if** $sendmessage == recvmessage$ **then**

23 $recvPktNum1 + +;$

24 $stream0.receiveStream.remove\,(recvEvent);$

25 if $(sendPktNum1 - recvPktNum1) * 100/sendPktNum1 > threshold$ **then**

26   **return** False;

27 return True;

4.6 Data Structures for Identified Communications

In the previous sections, I elaborate all the essential algorithms to identify the communications. The information of identified communications should be organized properly for the further presentation or visualization to 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

1   $communications \leftarrow Map\langle String, List\langle Communication \rangle\rangle;$
   $stream\_traces \leftarrow Map\langle String, List\langle Stream \rangle\rangle;$ **struct** {

2   Stream stream0   `// stream0 is from` $trace_0$ `of the dual-trace`

3   Stream stream1   `// stream1 is from` $trace_1$ `of the dual-trace`

4   DataMatch dataMatch

5   } *Communication*

6   **union** {

7   DataUnionMatch   unionMatch   `// For data union verification`

8   List $\langle$ EventMatch $\rangle$ eventMatchs   `// For data event verification`

9   } *DataMatch*

10   **struct** {

11   String sData1   `// send data union of endpoint1`

12   String rData1   `// receive data union of endpoint1,substring of sData2`

13   String sData2   `// send data union of endpoint2`

14   String rData2   `// receive data union of endpoint2,substring of sData1`

15   } *DataUnionMatch*

16   **struct** {

17   Event   event1   `// event1 is from enpoint1`

18   Event   event2   `// event2 is from enpoint2`

19   } *EventMatch*

20   **struct** {

21   Int   handle

22   List $\langle$ Event $\rangle$ openStream

23   List $\langle$ Event $\rangle$ closeStream

24   List $\langle$ Event $\rangle$ sendStream

25   List $\langle$ Event $\rangle$ receiveStream

26   } *Stream*

27   **struct** {

28   Int   stratline

29   Int   endline

30   Map $\langle$ String, String $\rangle$ inputs

31   Map $\langle$ String, String $\rangle$ outputs

32   } *Event*

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.[1] 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 Section4.2, 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.
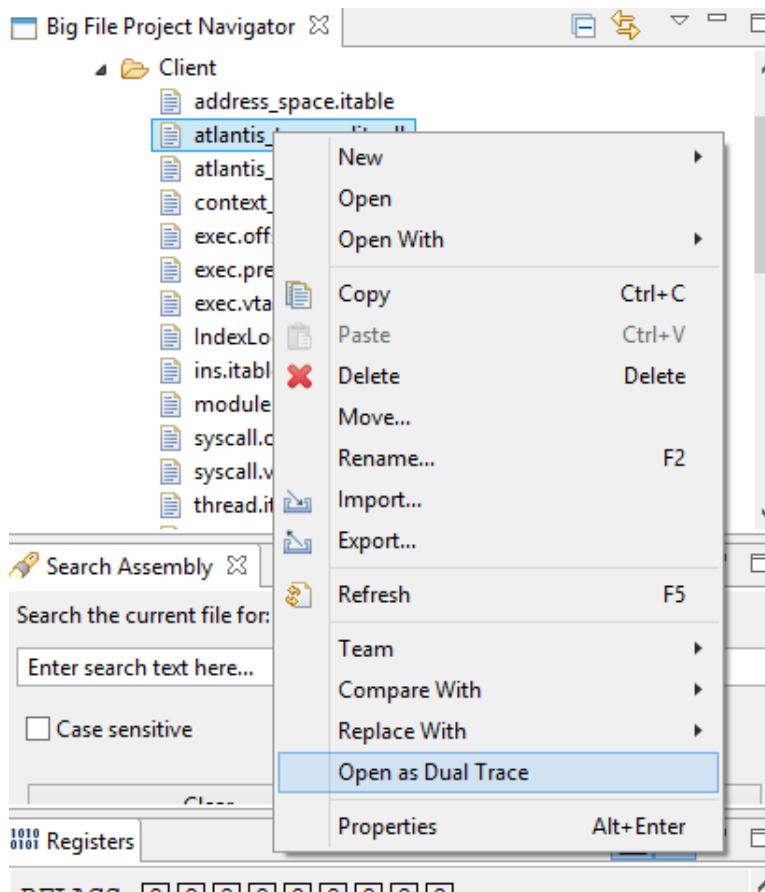
Figure 5.1: Menu Item for opening Dual_trace
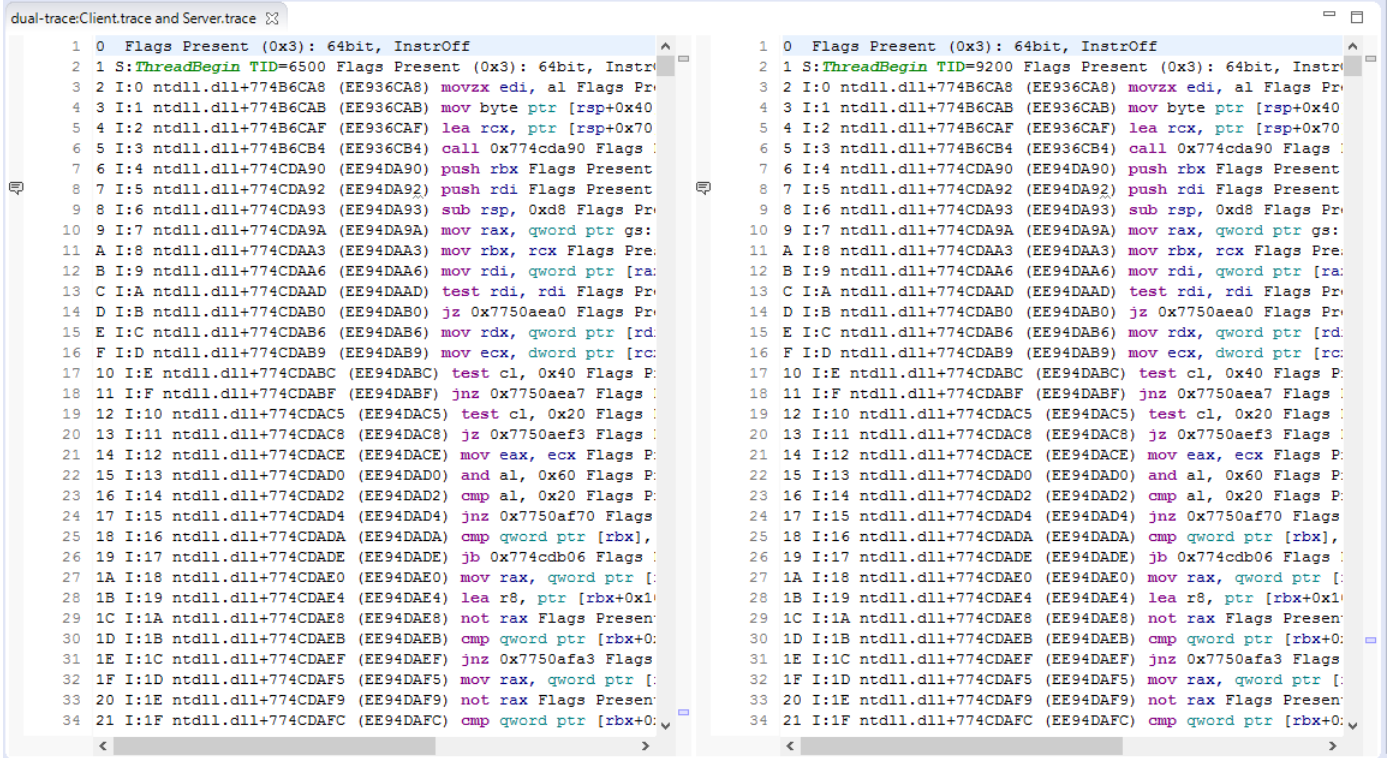
Figure 5.2: Parallel Editor View

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.
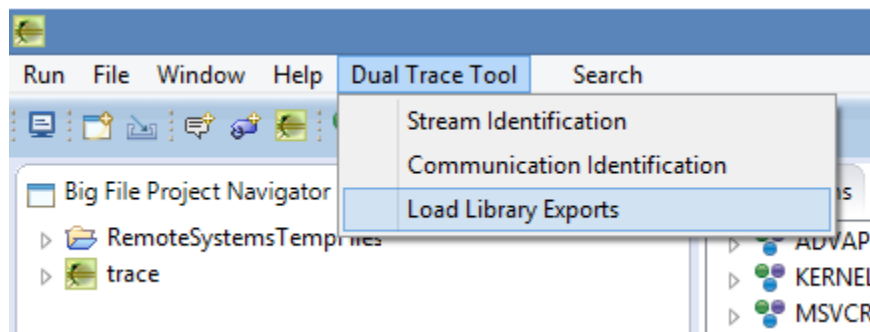
Figure 5.3: Dual_trace Tool Menu
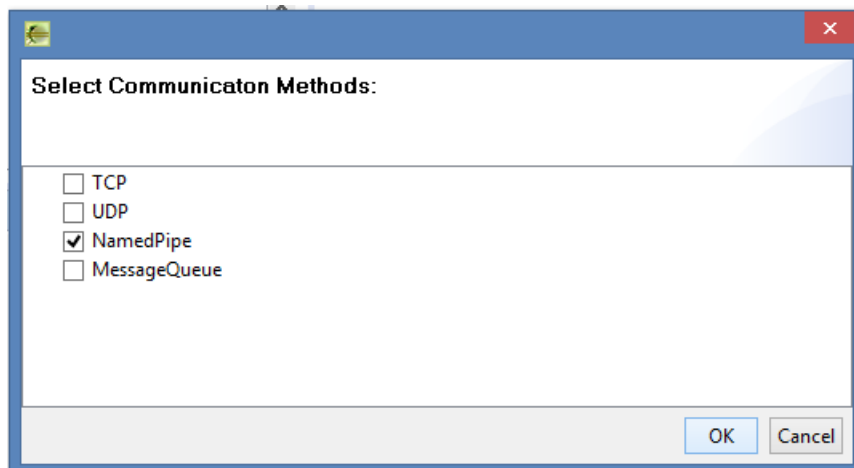
Figure 5.4: Prompt Dialog for Communication Selection

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.
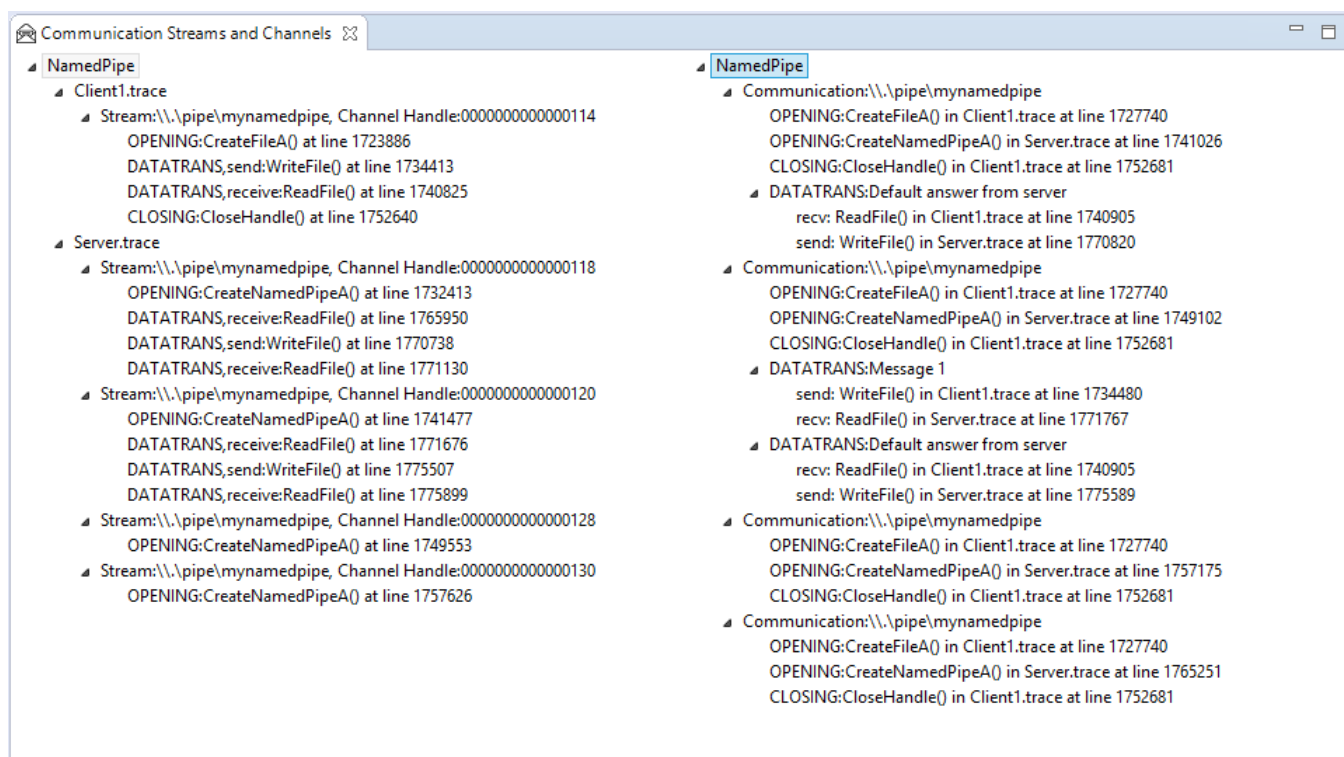
Figure 5.5: Communication View for Showing Identification Result

5.4   Identification Result View and Result Navigation

Atlantis is a analysis environment that has various views to allow user access to different informa-tion 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.
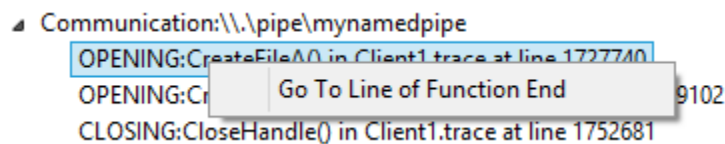
Figure 5.6: Right Click Menu on Event Entry

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 "commu-nication" entry. This provides the user the flexibility to get rid of the data that they don't care.
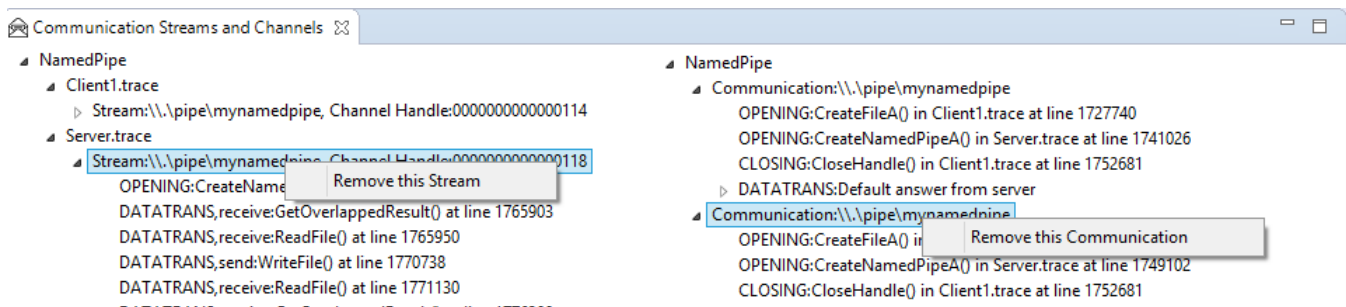
Figure 5.7: Right Click Menu on Event Entry

Chapter 6

Evaluation

This evaluation aimed to experimentally evaluate the model for communication analysis and the identification algorithms. The evaluation only focus on evaluating the correctness and efficiency of the model and algorithm. 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. In this section, I describe the design of the evaluation. Then the results of the experiment evaluation are provided and concluded.

6.0.1   Experiment Evaluation Design

Two experiments are conducted for this evaluation. All the programs in these two experiments were written in C++ and the source code can be found in SectionE. Our search partner DRDC provided the captured traces, the used .dll files and the source code of the programs for the experiments.

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.
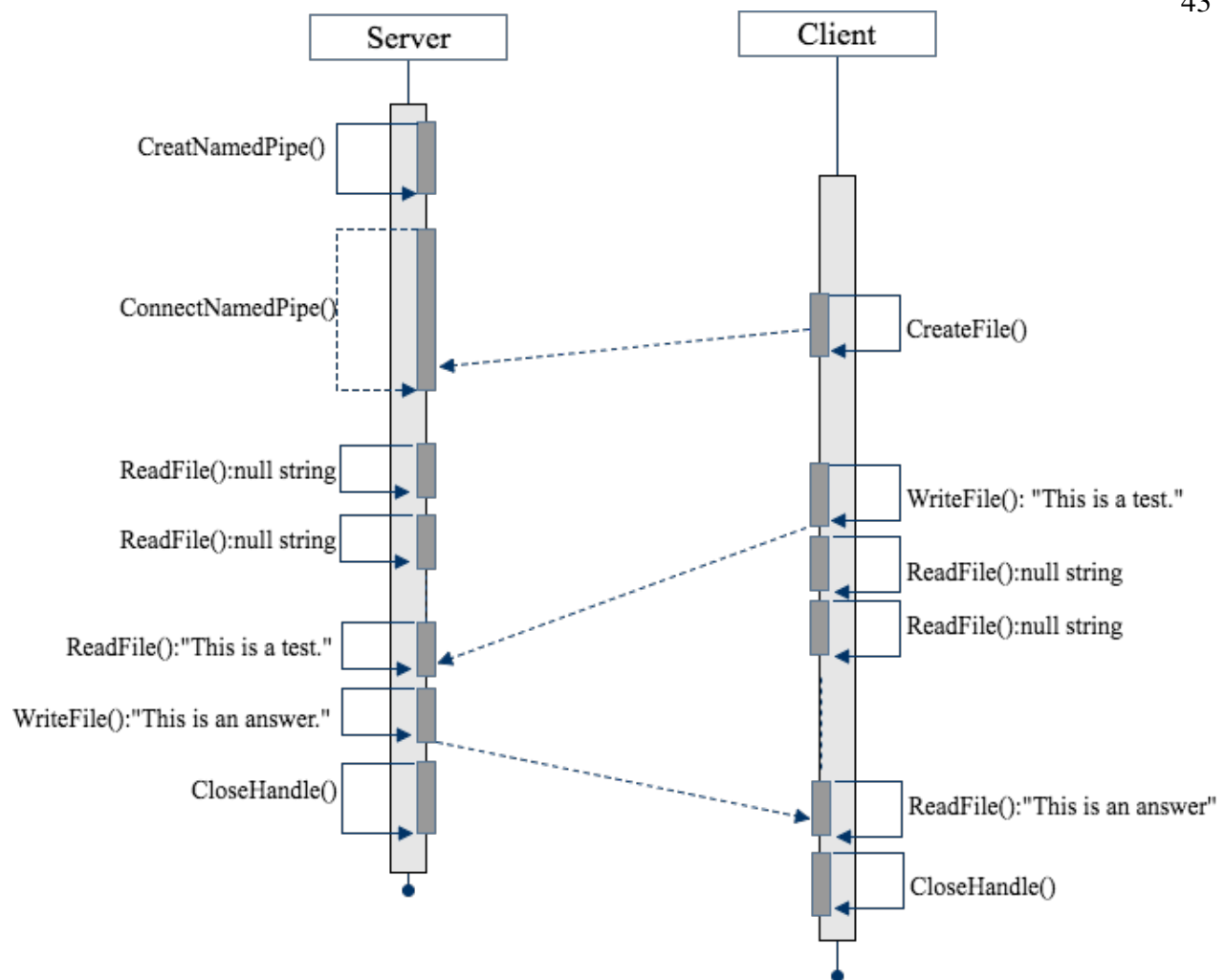
Figure 6.1: Sequence Diagram of Experiment 1

In the second experiment, one program was running as the Named pipe server, while four other programs as the Named pipe clients connected to this server. Those four clients (client 1, client 2, client 3 and client 4) used the identical program but run in sequence. Figure6.2 is the sequence diagram of the interaction among the server and clients. 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.

Figure 6.2: Sequence Diagram of Experiment 2

I used the implemented feature on Atlantis to conduct the dual_traces analysis for the experiments. "Communication identification" was conducted for all dual_traces. The result of these two experiments is discussed in next section.

6.0.2 Result and Conclusion

The identification result and the processing time of the identifications of each experiment are listed in Table6.1. All communications in the dual_traces are being successfully identified. However, dual to the channel id and transmitted data of the communication between server and client 1 was identical to those of the communication between server and client 2. There is one false negative error in exp2.1 and one in exp2.2 which is align to the explanation in Section4.5. The dual_traces being analysed is relatively small, so that the processing times of the identification are acceptable. Fortunately, according to the time complexity of the main algorithm of the identification which is O(N+M), the processing time will only grow linearly corresponding to the size of the dual_trace.

Table 6.1: Communication Identification Experiment Result

Dual_trace	Actual[1]	Success[2]	Error[3]	Processing Time(s)
exp1	1	1	0	0.0
exp2.1	1	1	1	0.0
exp2.2	1	1	1	0.0

[1] Actual communication number in this dual_trace

[2] Success identified communication number of this dual_trace

[2] Error identified communication number of this dual_trace, including false negative and false positive errors

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 experiment evaluation 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] 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.

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

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

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

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

```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": "na"
        }
      ]
   }
]
```

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

```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;
```

```c
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;
```

```c
                        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"
start "Client 3" NamedPipeClient.exe "Message 3"
start "Client 4" NamedPipeClient.exe "Message 4"
```