Communication Analysis of Programs through Assembly Level Execution Traces

by

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

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

MASTER OF SCIENCE

in the Department of Computer Science

by

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

Supervisory Committee

---

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

---

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

**Supervisory Committee**

---

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

---

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

## ABSTRACT

Understanding the communication between programs can help the software security engineers understand the behaviour of a system and detect the vulnerabilities of a system. Assembly level execution traces are used for analyzing the communications between programs for the two reasons: 1) lack of source code of the running programs, 2) assembly execution traces provide more accurate run time behaviour information about the system. In this thesis, I present a communication analysis approach using assembly level execution traces. I first defined the communication model in the context of trace analysis. Then I developed a process and the necessary algorithms to identify the communications from a dual_trace which consist of two assembly level execution traces. A prototype is developed for communication analysis. Finally, I conducted two experiments for communication analysis of interacting programs. These two experiments shows the usefulness of the designed communication analysis approach, the developed algorithms and the implemented prototype.

# Contents

# List of Tables

# List of Figures

# ACKNOWLEDGEMENTS

I would like to thank:

# DEDICATION

Just hoping this is useful!

# Chapter 1

# Introduction

Vulnerabilities in software enable the exploitation of the computer or system they are running on. Therefore, the emphasis placed on computer security particularly in the field of software vulnerabilities has increased dramatically. It's important for software developers to build secure applications. Unfortunately, building secure software is expensive. The vendors usually comply with their own quality assurance measures which focus on marketable concerns while leaving security to a lower priority or even worse, totally ignore it. Therefore, fully relying on the vendor of the software to secure your system and data is unpractical and risky.[**?**]

Software security review conducted by a third party is neccessary. One approach of software security review is software auditing. It is a process of analyzing the software in forms of source code or binary. This auditing can uncover some hard to reveal vulnerabilities which might be exploited by hackers. Identification of these security holes can save the users of the software from putting their sensitive data and business resources at risk.[**?**]

Most of the software vulnerabilities are stimulated by malicious data. So it is valuable to understand how this malicious data triggers the unexpected behaviors of the system. In most cases, this malicious data is injected by attackers into the system to trigger the exploitation. In some complex systems, several programs work together to provide service or functionality. In these situations, the malicious data might have passed through multiple components of the system and be modified before it reaches the vulnerable point and ultimately triggers an exploitable condition of the system. As a consequence, the flow of data throughout the system's different programs is considered to be one of the most important aspects to analyze during the security review.[**?**]

The data flow among various programs within a system or across different systems helps to understand how the system works as well as potentially disclose the vulnerabilities in a system. There are multiple mechanism to grab the data across programs. And the methods for obtaining

this data flow can affect the analysis results greatly.

In this research, I developed a method to identify communications between programs by analysing the assembly level execution traces. This method can guide the security engineers to investigate the communications of the programs in the circumstance that they have the captured execution traces and want to understand the interaction behaviour of the programs. The research is not specific for vulnerabilities detection but generalized for the comprehension of the interacting behaviour of two programs.

## 1.1 Motivation

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

### 1.1.1 Why Assembly Trace Analysis

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

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

The instrumentation of the tracing can be done at various levels of granularity, such as programming language or machine language instructions. The access to a software can be divided into five categories, with variations: source only, binary only, both source and binary access, checked build, strict black box. Only having the binary is common when performing vulnerability research on closed-source commercial software[**?**]. In this case, assembly level tracing is the only option to do a security review of the software.

On the other hand, the binary code is what is running on the system, binary tracing is more representative of actual situation than the source code. Some bugs might appear because of a com-

pilation problem or because the compiler optimized away some code that is necessary to make the system secure. The piece of code listed below is an example in which the line of code re-setting the password before the program end would be optimized away by the compliers if they implement the dead store elimination[**?**]. For example, with the -fdse option, the GNU Compiler Collection(GCC) will perform the dead store elimination and -fdse is enabled by default at -O and higher optimization level [**?**]. This made the user's password stays in memory, which is considered as potential security risk. However, looking at the source code does not reveal the problem.

Listing 1.1: Password Fetching Example

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

### 1.1.2 Why Communication Analysis with Assembly Traces

Programs nowadays do not always work isolated. The communication and interaction between programs affect the behaviour of the system. Without knowing how a program works with others, an analysis of the isolated execution trace on a single computer is usually futile. Data flow tracing between programs is essential to review both the design and implementation of the software.

Many network sniffer, such as Wireshark[**?**] and Tcpdump[**?**], can help to capture the data flow across the network. However, this method is insufficient because security problems can occur even if the information sent is correct. Therefore, analysing the communications with transmitted data in instruction and memory access level is a solid way to evaluation the security of a system.

Shameng Wen et al. argued that fuzz testing and symbolic execution are widely applied to detect vulnerabilities in network protocol implementations. Their work focuses on designing a

model which guides the symbolic execution for the fuzz testing [**?**] but ignoring the analysis of the output, which is the execution traces. Furthermore, their work focused only on the network protocol implementation and is not generalized to all communications.

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

Overall, communication analysis with assembly traces is a way to learn how the work is performed by the system.

## 1.2 Research Goal

The goal of this research is to design a method for communication analysis using the execution traces of the interacting programs. This method should be general enough for all message based communication analysis between programs regardless of their programming language, host operating system or selected execution tracer.

## 1.3 Research Process

Figure 1.1 shows the overview of my research process with three abstracted stages. However, The approach of this research is not a forthright process. Instead, it is a back and forth one, for example the implementation changed several times with the changes of the model, and the models was modified based on the understanding throughout the implementation.

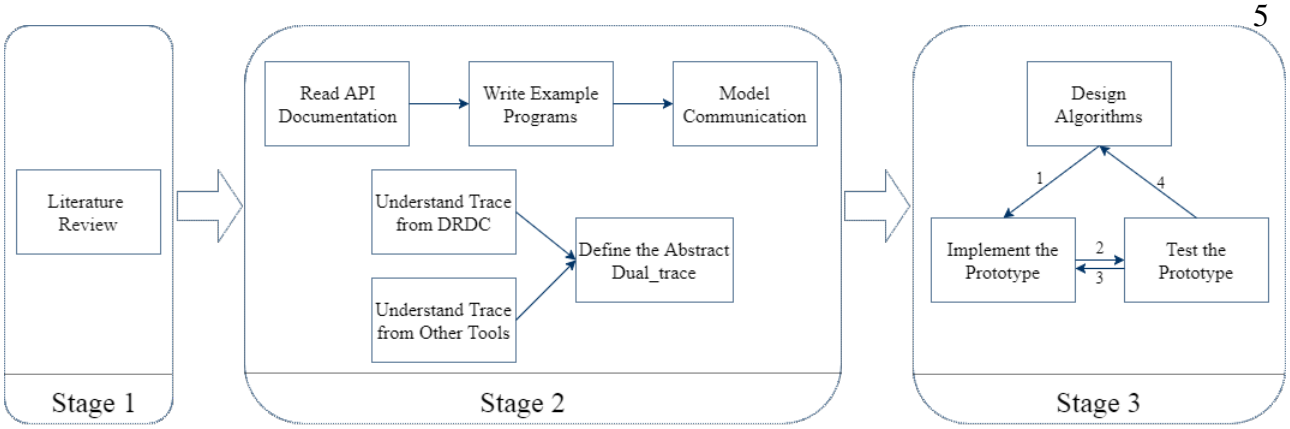



Figure 1.1: Research Approach overview

This research requires background knowledge of software security and vulnerabilities. I acquired the background knowledge basically from literature review. It helped me grab the essential concepts of software vulnerabilities and their categories, understand some facilities for vulnerabilities detection and software maintenance in the perspective of security. After that, I was convinced that communication analysis in assembly trace level would benefit software security engineers to understand the behaviour of software and detect software vulnerabilities.

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

The abstract assembly trace definition was build on the generalization of the trace format provided by our research partner, DRDC. I don't have the access to their home-made assembly tracer which is based on PIN[**?**]. Fortunately, they provideed me with a comprehensive document about the format of the captured trace and example traces. With these, I grasped the constructive view of the assembly execution trace. Further more, some other tools can also capture the required information in assembly level for communication analysis. This supports the generalization of the trace definition and the abstraction of the dual_trace.

The implementation of the prototype and the communication analysis algorithms were developed in parallel. The high level communication identification algorithm and the specific algorithms for named pipe communication methods were abstracted based on the implementation, while the others are developed theoretically. Two experiments are designed to test this analysis method, the prototype and some algorithms.

## 1.4 Contributions

The main contributions of this work can be summarized as:

- **Communication Model:** The communication model in this thesis is an abstract model of the communications between two programs. It was abstracted from the understanding of several communication methods and is general for other communication methods that are not mentioned in this thesis.
- **Dual_trace Formalization:** By understanding the assembly level execution traces, a dual_trace was formalized to describe the information that needed for communication analysis. This model doesn't specify the format of the execution traces but defines what information is necessary to be contained to fulfil the analysis requirement. All execution traces that comply with this definition can be used for the analysis.
- **Communication Analysis Approach:** The overall approach to identify the communications in the dual_trace is designed. Algorithms were developed for the steps in this approach regarding some communication methods or communication types.
- **Prototype:** A prototype is designed and implemented on Atlantis. This prototype demonstrate that the communication analysis approach is feasible. It is a unique tool for the security engineers to analyze the communications of programs via assembly execution trace analysis.

## 1.5 Thesis Organization

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

Chapter 3 describes the model of the communication between two programs. This model defines the communication in the context of trace analysis and discusses the properties of the communications.

In Chapter 4, I first present the abstract dual_trace formulation. Based on this formulation, I describe the communication analysis process and the essential algorithms.

In chapter 5, I present the implementation of a dual_trace communication analysis prototype.

In chapter 6, I present two experiments of communication analysis with dual_traces using the implemented prototype. Notably, the result shows the communications are correctly identified.

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

# Chapter 2

# Background

In this chapter, I summarize the background related to this work. First I generally describe what is a software vulnerability. Second, I discuss the categorization of communications among programs. Third, I introduce some tools for assembly level program debugging and analysis. Finally I introduce Atlantis, the existing assembly level execution trace analysis environment, on which the prototype of this work is based.

## 2.1 Software Vulnerability

Software vulnerability detection is one of the use cases of the communication analysis with assembly level execution traces. Vulnerabilities, from the point of view of software security, are specific flaws or oversights in a program that can be exploited by attackers to do something malicious, such as modify sensitive information, disrupt or destroy a system, or take control of a computer system or program[**?**]. They are considered to be a subset of bugs. Input and data flow, interface and exceptional condition handling are where vulnerabilities most likely to surface in software. Memory corruption is one of the most common vulnerabilities. The awareness of these would make the security auditing and vulnerabilities detection have more clear focus.

## 2.2 Program Communications Categories

Programs can communicate with each other via diverse mechanisms. The communication happens among processes is known as inter-process communication. This refers to the mechanisms an operating system provides a process to share data with each other. It includes methods such as signal, socket, message queue, shared memory and so on [**?**]. These communications can happen

over network or inside a device. Based on their reliability, the communication methods can be divided into two categories: reliable communication and unreliable communication. In this work, both communication methods are covered. However, I only discuss message based communication methods.

## 2.3 Program Execution Tracing in Assembly Level

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

There are many tools that can trace a running program in assembly instruction level. IDA pro [**?**] 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 OllyDbg[**?**] also provide the debugging and tracing functionality in assembly level.

## 2.4 Atlantis

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

# Chapter 3

# Communication Modeling

In this chapter, I model the communication of two running programs from the trace analysis point of view. The modeling is based on the investigation of some common used communication methods. But the detail of the communication methods will be discussed later in the algorithm and implementation chapters. This chapter only present the abstract communication model regarding to the two communication categories: reliable and unreliable communications.

## 3.1 Communication Methods Categorization

In terms of their reliability of data transmission, there are two types of communication: reliable and unreliable. A reliable communication guarantees the data being sent by one endpoint through the channel is always received losslessly and in the same order in the other endpoint. On contrast, 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 the unreliable communications is that the packets being sent are always arrived as the origin packet, no data re-segmentation would happen. An endpoint is an instance in a program at which a stream of data is sent or a stream of data is received or both (e.g. socket handle of TCP or a file handle of the named pipe). A channel is a conduit connected two endpoints through which data can be sent and received. Table3.1 gives examples of communication methods fall in these two categories.

Table 3.1: Communication Method Examples in Two Categories

| **Reliable Communication** | **Unreliable Communication** |
|---|---|
| Named Pipes | Message Queue |
| TCP | 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 is considered for a new communication. The way that I define the communication was leading to the communication analysis. So the definition is not about how the communication works but what it looks like. There are many communication methods in the real world and they are compatible to this communication definition.

### 3.2.1 Communication Definition

In the context of a dual_trace, a communication is a sequence of data transmitted of two endpoints through a communication channel. I, therefore, defined a communication $c$ as a triplet:

$c =< ch, e_0, e_1 >$

where $e_0$ and $e_1$ are endpoints while $ch$ is the communication channel (e.g. a named piped located at /tmp/pipe).

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

$e =< handle, d_r, d_s >$

where $handle$ is the handle identifier of the endpoint, $d_r$ is the data stream received and $d_s$ is the data stream sent. A data stream is a sequence of sent packets or a sequence of received packets. Each packet $pk$ contains data that is being sent or received (its payload). Hence, we can define a data stream $d$ as a sequence of $n$ packets:

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

Note: This is the sequence of packets as seen from the endpoint and might be different than the sequence of packets seen in the other endpoint, specially where there is packet reordering, loss or duplication.

Each packet $pk$ has two attributes:

- *Relative time (it was sent or received):* In a trace, we do not have absolute time for an event. However, we know when an event (i.e. open, close, sending or receiving a packet)

has happened with respect to another event. I use the notation

$time(pk)$

to denote this relative time. Hence,

if $i < j$ , then $time(pk_i) < time(pk_j)$

- *Payload:* Each packet has a payload (the data being sent or received). I use the notation

  $pl(pk)$

  to denote this payload.

### 3.2.2 Communication Properties

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

#### Properties of reliable communication

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

For a given data stream, we define the data in this stream as the concatenation of the payload of all the packets in this stream, in the same order, and denote it as $data(d)$.

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

- *Content Preservation:*

  For a communication, the received data of an endpoint should always be a prefix of (potentially equal to) the sent data of the other:

  for $c =< ch, < h_0, dr_0, ds_0 >, < h_1, dr_1, ds_1 >>$,$data(dr_0)$ is a prefix of $data(ds_1)$ and $data(dr_1)$ is a prefix of $data(ds_0)$.

- *Timing Preservation:*

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

  for a sent data stream of size $m$, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size $n$, $dr =< pkr_1, pkr_2, ...pkr_n >$, for any $k \in 1..n$, there must exist $j \in 1..m$ such that $pks_j$ was sent before $pkr_k$ was received:

$time(pks_j) < time(pkr_k)$

and

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

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

**Properties of unreliable communication**

In unreliable communication the properties are not concerned in the concatenation of packets. Instead, each packet is treated as independent of each other.

- *Content Preservation:*

  A packet that is received should have been sent:

  for a sent data stream of size $m$, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size $n$, $dr =< pkr_1, pkr_2, ...pkr_n >$, for any $pkr_j \in dr$ there must exist $pks_i \in ds$, we will say that the $pkr_j$ is the matched packet of $pks_i$, and vice-versa, $pks_i$ is the matched packet of $pkr_j$, hence $match(pkr_j) = pks_i$ and $match(pks_i) = pkr_j$.

- *Timing Preservation:*

  At any given point in time, packets can only be received if they have been sent:

  for a sent data stream of size $m$, $ds =< pks_1, pks_2, ...pks_m >$ that is received in data stream of size $n$, $dr =< pkr_1, pkr_2, ...pkr_n >$, for any $k \in 1..n$, $time(match(pkr_j)) < time(pkr_j)$.

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

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

Figure3.1 is an example of the reliable communication.

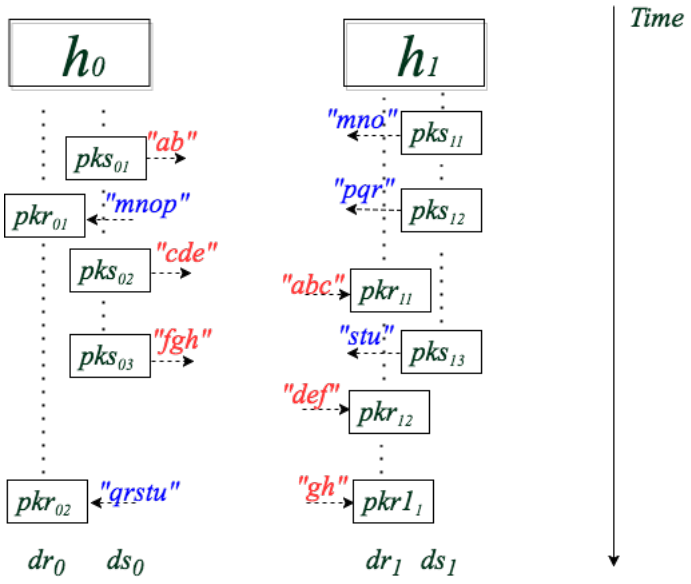



Figure 3.1: Example of Reliable Communication

In this example, the payloads of the packets are:
$pl(pks_{01}) =$ "$ab$", $pl(pks_{02}) =$ "$cde$", $pl(pks_{03}) =$ "$fgh$";
$pl(pkr_{11}) =$ "$abc$", $pl(pkr_{12}) =$ "$def$", $pl(pkr_{13}) =$ "$gh$" .
in one direction and
$pl(pks_{11}) =$ "$mno$", $pl(pks_{12}) =$ "$pqr$", $pl(pks_{13}) =$ "$stu$";
$pl(pkr_{01}) =$ "$mnop$", $pl(pkr_{02}) =$ "$qrstu$".
on the other direction.
Their properties:
$pl(pks_{01}) \cdot pl(pks_{02}) \cdot pl(pks_{03}) = pl(pkr_{11}) \cdot pl(pkr_{12}) \cdot pl(pkr_{13}) =$ "$abcdefgh$"
and
$pl(pks_{11}) \cdot pl(pks_{12}) \cdot pl(pks_{13}) = pl(pkr_{01}) \cdot pl(pkr_{02}) =$ "$mnopqrstu$".
satisfy the content preservation.
The relative time relationship of the packets are:
$time(pks_{01}) < time(pks_{02}) < time(pkr_{11}) < time(pks_{03}) < time(pkr_{12}) < time(pkr_{13})$;
$time(pks_{11}) < time(pks_{12}) < time(pkr_{01}) < time(pks_{13}) < time(pkr_{02})$.
The fact that
$pl(pkr_{01}) =$ "$mnop$" is the prefix of $pl(pks_{11}) \cdot pl(pks_{12}) =$ "$mnopqr$",
$pl(pkr_{01}) \cdot pl(pkr_{02}) =$ "$mnopqrstu$" is the prefix of (in this case is identical to ) $pl(pks_{11}) \cdot pl(pks_{12}) \cdot pl(pks_{13}) =$ "$mnopqrstu$",
$pl(pkr_{11}) =$ "$abc$" is the prefix of $pl(pks_{01} \cdot pl(pks_{02}) =$ "$abcde$",

$pl(pkr_{11}) \cdot pl(pkr_{12}) =$ "$abcdef$" and $pl(pkr_{11}) \cdot pl(pkr_{12}) \cdot pl(pkr_{13}) =$ "$abcdefgh$" are the prefix of $pl(pks_{01}) \cdot pl(pks_{02}) \cdot pl(pks_{03}) =$ "$abcdefgh$"

satisfy the timing preservation.

Figure3.2 is an example of the unreliable communication.

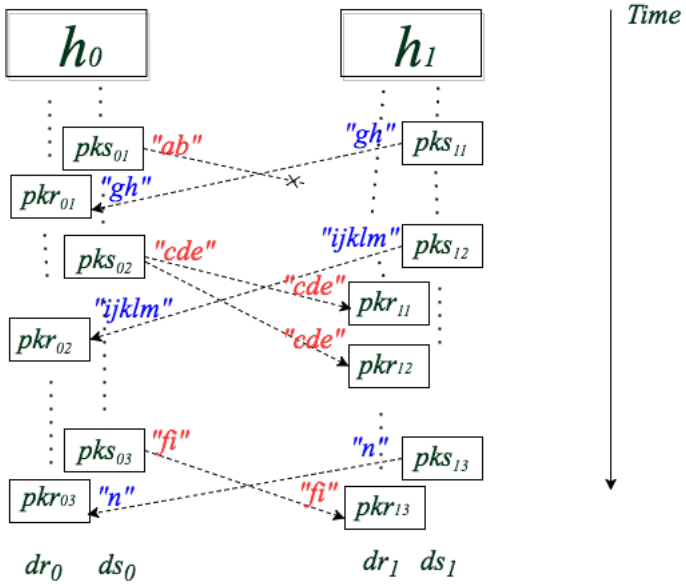



Figure 3.2: Example of Unreliable Communication

In this example, the content preservation of the unreliable communication are satisfied since:

$pkr_{11} = pks_{02} =$ "$cde$";

$pkr_{12} = pks_{02} =$ "$cde$";

$pkr_{13} = pks_{03} =$ "$fi$";

$pkr_{01} = pks_{11} =$ "$gh$";

$pkr_{02} = pks_{12} =$ "$ijklm$";

$pkr_{03} = pks_{13} =$ "$n$".

while the timing preservation of the unreliable communication are satisfied since:

$time(pkr_{11}) > time(pks_{02})$;

$time(pkr_{12}) > time(pks_{02})$;

$time(pkr_{13}) > time(pks_{03})$;

$time(pkr_{01}) > time(pks_{11})$;

$time(pkr_{02}) > time(pks_{12})$;

$time(pkr_{03}) > time(pks_{13})$;

# Chapter 4

# Communication Analysis

I defined a message transferring communication between two programs in Chapter 3. The goal of this thesis is to develop a method to identify the communications from the dual_trace. A dual_trace is a pair of assembly level execution traces of two interacting programs. In this chapter, I discuss the characteristics of the assembly level execution trace. And then formalize the dual_trace and the execution traces in the. For all the traces comply with this abstract trace formalization, the analysis approach presented in this chapter can be applied.

The process of the communication analysis is shown in Figure 4.1. It takes the two traces in the dual_trace as input and outputs the identified communications. In this overview figure, there are four components. The function call event reconstruction component will analyze the traces and try to reconstruct all function calls of the functions in the function descriptor. These two sequence of events of these two traces will then flow into the stream extraction component separately. In each event sequence, the events might be trigger by different endpoints of different communications. I consider all the events triggered by the same endpoint as a stream. The stream extraction component will extract two sets of streams. After that, the stream matching component will take both of the stream sets as input and try to match them by their channel identifiers and output the potential identified communications. Finally, the data verification component will verify each communication satisfying the communication data preservation. Algorithms are designed separately for each component. Details about each elements and components of this overall process will be discussed in the following sections.

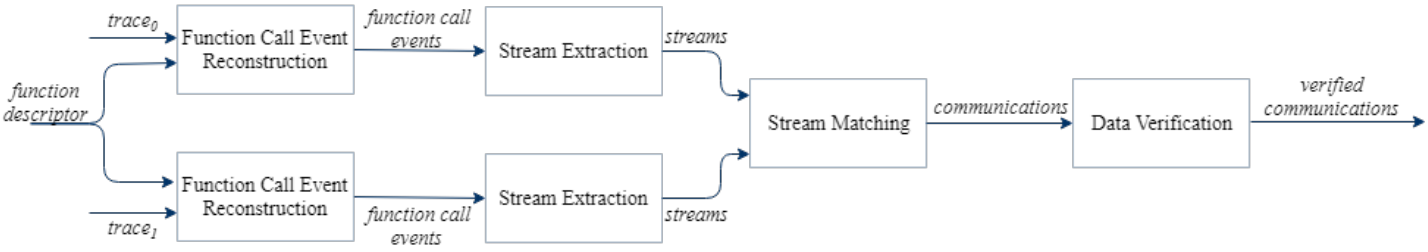



Figure 4.1: Process of the Communication Analysis through a Dual_trace

## 4.1 Dual_Trace

In this section, I formalize a dual_trace. All traces aligning with this formalization can be used as the input of the analysis process shown in Figure 4.1. A dual_trace consists of two assembly level execution traces of two interacting programs. There is no timing information of these two traces which means we don't know the timing relationship of the events of one trace with respect to the other. However, the captured instructions in a trace are ordered in execution sequence.

An execution trace consists of a sequence of instruction lines. Each instruction line contains the executed instruction, the changed memory, the changed registers and the execution information.

A dual_trace is formalized as :

$dual\_trace = \{trace_0, trace_1\}$

where $trace_0$ and $trace_1$ are two assembly execution traces.

A trace is a sequence of executed instruction lines:

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

Each instruction line, $l$, is a tuple:

$l =< ins, mch, rch, exetype, syscallInfo >$

where $ins$ is the instruction, $mch$ is the memory changes, $rch$ is the register changes, $exetype$ is the execution type which can be instruction, system call entry, system call exit, and other types which are not concerned in this work, $syscallInfo =< exeName, funcName >$ only appears when $exetype$ is system call entry or system call exit. $exeName$ is the executable file name (e.g. .dll and .exe), while $funcName$ is the name of a system function in this executable file.

Figure 4.2 is an example of a piece of execution trace complying to this definition.

| Line | Instruction | Memory Changes | Registster Changes | Execution Type | System Call Info |
|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... |
| $l_{1499}$ | lea rcx, ptr [rip+0x3cfe0] | | ECX F8A0AAA8 | Instruction | |
| $l_{1500}$ | xor r9d, r9d | | | Instruction | |
| $l_{1501}$ | mov edx, 0xc0000000 | | EDX C0000000 | Instruction | |
| $l_{1502}$ | mov dword ptr [rsp+0x20], r8d | 00000000001DF490 00000003 00000000 | RSP 00000000 001DF470 | Instruction | |
| $l_{1503}$ | call qword ptr [rip+0x3a97d] | 00000000001DF470 000007FE F89CDADB | RSP 00000000 001DF468 | Instruction | |
| $l_{1504}$ | mov qword ptr [rsp+0x8], rbx | 00000000001DF470 00000000 00000001 | EBX 00000001 | System Call Entry | kernel32.dll+ CreateFileW |
| $l_{1505}$ | mov qword ptr [rsp+0x10], rbp | 00000000001DF478 00000000 00000000 | EBP 00000000 ESP 001DF468 | Instruction | |
| ... | ... | ... | ... | ... | ... |
| $l_{2057}$ | add rsp, 0x20 | | ESP 001DF2A0 | Instruction | |
| $l_{2058}$ | pop rbx | | RBX 00000000 001DF3E8 RSP 00000000 001DF2A8 | Instruction | |
| $l_{2059}$ | ret | | RSP 00000000 001DF2B0 | System Call Exit | kernel32.dll+ CreateFileW |
| $l_{2060}$ | mov eax, dword ptr [rsp+0x54] | | EAX 00000000 | Instruction | |
| ... | ... | ... | ... | ... | ... |

Figure 4.2: An example trace

## 4.2 Functions Descriptors

There would be lots of function calls in an execution trace. However, most of them are not of interest. I am only concerned with the function call events of a specific communication method. To be able to identify and reconstruct the function calls of interest, the function descriptions are required. I define the function descriptor as:

$cdesc = \{fdesc_1, fdesc_2, ..., fdesc_p\}$

Each element, $fdesc$, is a function description and can be defined as:

$fdesc = \{name, type, inparamdesc, outparamdesc\}$

where, $name$ is the function name, $type$ is the function type which can be one of the four types: $open$, $close$, $send$ and $receive$. $inparamdesc$ is the input parameter descriptions illustrating how the registers and memory contents map to a list of parameters of interest(you might not care for all parameters) of a given function call and $outparamdesc$ is the output parameter descriptions similar to the input parameter descriptions.

Table4.1 is an example of a function description. In this example, the function name is $ReadFile$, it is a function for data receiving, so that function type is $receive$. The input parameter description has one concerned parameter, $Handle$, while the output parameter description has two parameters, $RecvBuffer$ and $MessageLength$. $Handle$ is a parameter which is a value stored in the register RCX. The $RecvBuffer$ is an address for the input message stored in the register RAX. The $MessageLength$ is a output value stored in register R9. The value of the input parameters can be retrieved from the memory state on the function call instruction line while the value of the output parameters can be retrieved from the memory state on the function return instruction line. If a parameter is an address instead of value, the address should be retrieved first, then the retrieved address should be used to find the buffer content in the memory state. The function description requires the understanding of the calling convention of the operating system. The Microsoft x64

calling convention can be found in Appendix A. More examples of communication method descriptions will be given in Chapter 5.

Table 4.1: An example of a function description

| **Name** | **Type** | **Input Parameter Description** | | | **Output Parameter Description** | | |
|---|---|---|---|---|---|---|---|
| | | **Name** | **Register** | **Addr/Val** | **Name** | **Register** | **Addr/Val** |
| ReadFile | receive | Handle | RCX | Value | RecvBuffer | RDX | Addr |
| | | | | | MessageLength | R9 | Val |

## 4.3 Function Call Event Reconstruction Algorithm

In last two sections, I formalized the assembly execution trace and defined the function descriptor of a communication method. The function descriptor helps to locate the function calls and retrieve the parameters of interest from an execution trace. These function calls contain the information of a communication, such as the channel identifier, the packets sent or received, etc. Before any communication can be identified, the function calls of that communication method have to be reconstructed first.

In this section, I define the function call event and present the algorithm to reconstruct the function call events from an assembly execution trace.

With the function descriptor and the execution trace as input, the function call event reconstruction algorithm identifies the function call entry instruction line and reconstructs the input parameters from the memory state of that line. Then it identifies the function call exit line of the corresponding function call and reconstructs the output parameters from the memory state of the function exit line. After iterating the whole execution trace, the algorithm outputs a sequence of function call events of length $m$ which can be defined as:

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

A function call event $ev$ in $etr$ is defined as a tuple:

$ev =< funN, inparams, outparams, type >$

where $funN$ is the function name, $inparas$ includes all the input parameters with the parameter name and value, $outparas$ includes all the output parameters, and $type$ is the event type which inherit from the function description and can be one of the four types: $open$, $send$, $receive$ and $close$.

Note: If the parameter is an address, the parameter's value is the string from the buffer pointed by that address instead of the buffer address.

An example of a sequence of function call events as the output of this algorithm is shown in Listing4.1.

Listing 4.1: Example of $etr$

```
{funN:CreateNamedPipe, type:open,inparams:{Handle:18, FileName:mypipe}, outparams:{}},
{funN:CreateNamedPipe, type:open, inparams:{Handle:27, FileName:Apipe}, outparams:{}},
{funN:WriteFile, type:send, inparams:{Handle:27, SendBuf:Message1}, outparams:{MessageLen:9}},
{funN:WriteFile, type:send, inparams:{Handle:27, SendBuf:Message2}, outparams:{MessageLen:9}},
{funN:ReadFile, type:receive, inparams:{Handle:27}, outparams:{RecvBuf:Message3, MessageLen:9}},
{funN:CloseHandle, type:close, inparams:{Handle:27}, outparams:{}},
{funN:CloseHandle, type:close, inparams:{Handle:18}, outparams:{}}
```

Algorithm 1 presents the presudo code for the function call event reconstruction algorithm. This algorithm is designed to reconstruct the function call events for one communication method. If multiple communication methods are being investigated, this algorithm can be run multiple times to analysis each of them. Since there are usually a small number of functions of interest for a communication method compared to the instruction line number in the execution trace, the time complexity of this algorithm is $O(N)$ , $N$ is the instruction line number of the trace.

**Algorithm 1: Function Event Reconstruction Algorithm**

```
/* trace is the assembly execution trace with a sequence of instruction lines: (l_1, l_2, ..., l_n),
   cdesc is the function descriptor contains a set of function descriptions:
   fdesc_1, fdesc_2, ..., fdesc_p, etr is a sequence of function call events                    */
Input: trace, cdesc
Output: etr
1  etr ← ∅
2  i ← 1
   /* Emulate the Execute of each instruction line of the trace                                */
3  while i ≤ n do
4      l ← trace[i]
5      i ← i + 1
6      Execute the instruction of l
7      for fdesc ∈ cdesc do
8          if l is a call to the function described by fdes then
9              Create an new function call event ev
10             ev.funN ← fdesc.name
11             ev.type ← fdesc.type
12             Get the input parameters from the memory state and append them to ev.inparams
13             while i ≤ n do
14                 l ← trace[i]
15                 i ← i + 1
16                 Execute the instruction of l
17                 if l is a exit of the function described by fdes then
18                     Get the output parameters from the memory state and append them to ev.outparams
19                     Break the inner while loop
20             etr.append(ev)
21             Break the For loop
22 return etr
```

## 4.4 Channel Open Mechanisms

The channel open mechanism affect the stream extraction and stream matching strategy. So I discuss them before presenting those algorithms. The channel open mechanism of named pipe and message queue is relatively simple. In the Windows implementation, only one function call is related to the handle identification of the stream. However, for TCP and UDP the mechanism is complicated.

In all communication methods, all operations such as packet send and receive use a handle as an identifier to bind itself to an endpoint. This handle is generated or returned by a channel open function call and will be assigned to an input parameter for all other related function calls to indicate the corresponding endpoint. However, in other communication methods, the handles might have other names, such as file handle for Named Pipe or socket(sometime called socket

handle) for UDP and TCP. All of these are essentially equivalent.

A handle is a unique identifier among all open endpoints. An open endpoint is one that can still be used for data transfer. For example, if there are ten endpoints opened for communications, the handles for all these ten endpoints are different. However, if any of these endpoint is closed, its handle can be reused for any other new created endpoints. Since the handle is the unique identifier for an endpoint and its related events, we need to know it to identify an endpoint and its corresponding function call events.

Moreover, since two endpoints(one from each trace) are connected to a channel for communication, each endpoint has to know the identifier of the channel to connect to it. This channel identifier is usually given to the endpoint in the channel open function calls. The endpoint will remember this channel and know where the data should be sent to and received from. Therefore, to identify a communication, the channel identifier given to an endpoint needs to be found during the channel open stage.

In the following subsections, I will explain how different communication methods open their channels for communication.

### 4.4.1 Named Pipe Channel Open Mechanisms

For the Named Pipe communication method, 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 of a named pipe returns the file handle of that pipe. So the identification of the stream only needs to identify the pipe creation function call on the server side and the pipe connection function call on the client side. The handle returned by these function calls will be used later when data is being sent or received to a specified pipe. In a Named pipe, the file is used as the media of the channel, so the identifier in this case is the file name. Figure 4.3 exemplifies the channel set up process for a Named Pipe communication in Windows.

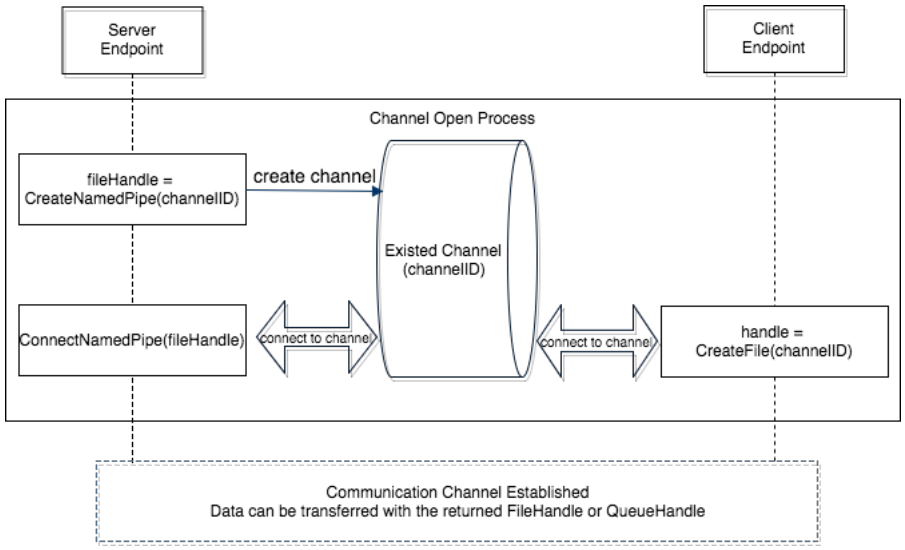



Figure 4.3: Channel Open Process for a Named Pipe in Windows

### 4.4.2 Message Queue Channel Open Mechanisms

For the Message Queue communication method, the endpoints of the communication can create the queue or use the existing one. However, both endpoints have to open the queue before they access it. The handle returned by the open queue function will be used later when messages are being sent or received to indicate the corresponding endpoint. The queue name as the input for the open queue function is the identifier of the channel. Figure 4.4 exemplifies the channel set up process for a Message Queue communication in Windows.

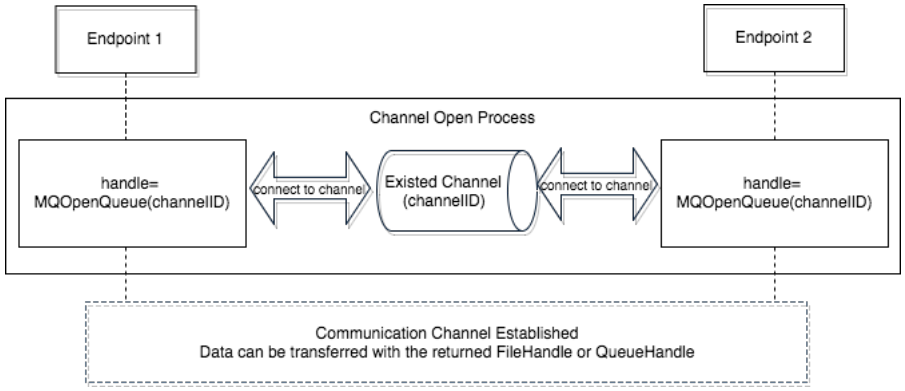



Figure 4.4: Channel Open Process for a Message Queue in Windows

### 4.4.3 UDP and TCP Channel Open Mechanisms

For the UDP and TCP communication methods, the communication channel is set up by both endpoints. The socket create function should be called to create their own socket on both endpoints. After the socket handles are created, the server endpoint binds the socket to its service address and port by calling the socket bind function. Then the server endpoint calls the listening function to accept the client connection. The client will call the connection function to connect to the server. When the listening function call returns 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. In this case, the server's address and port are considered to be the identifier of the channel. Fig. 4.5 exemplifies the channel open process for TCP and UDP in Windows.

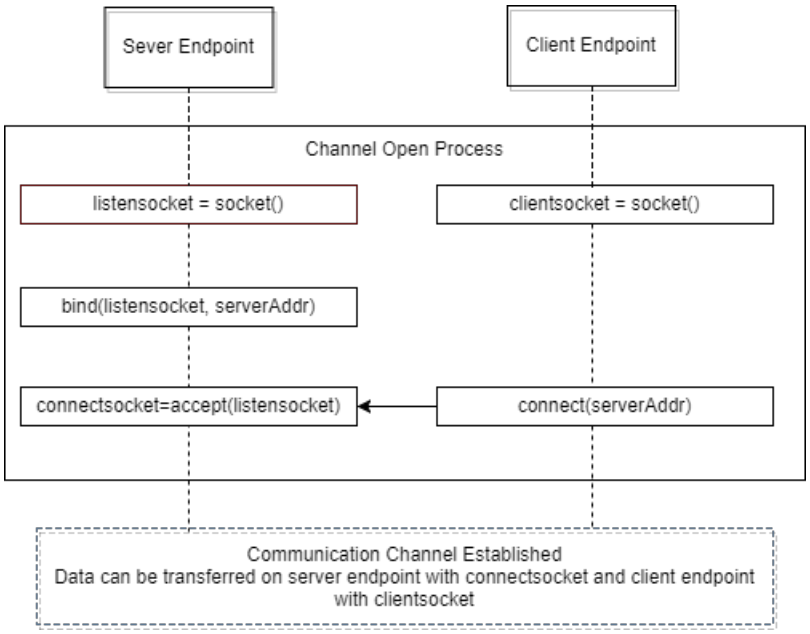



Figure 4.5: Channel Open Model for TCP and UDP in Windows

## 4.5 Stream Extraction Algorithm

The sequence of function call events output by the function call event reconstruction algorithm may belong to different endpoints. We need to further split these events for each endpoint. Each

subset of these events belonging to an endpoint is considered to be a stream. There are four types of events in each stream: open, send, receive and close. Hence, we can further divide a stream into sub streams which are called open stream, send stream, receive stream, and close stream. There will be only one type of events in each of these streams. Since a stream corresponds to an endpoint and an endpoint is connected to a channel, it is necessary to know the endpoint handle and the channel identifier corresponding to this stream.

A stream is formally defined as a tuple:

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

where $handle$ is the handle of the endpoint, $channelId$ is the identifier of the channel the endpoint of this stream is connected to, $so$ is the open stream, $ss$ is the send stream, $sr$ is the receive stream, $sc$ is the close stream.

The sub streams $so$, $ss$, $sr$, $sc$ are sequences of events, each sub stream $sx$ is defined as:

$sx = (ev_1, ev_2, ..., ev_p)$

Note: The event numbering of in this sub streams is different from the original sequence of event. For example, $ev_1$ in $s$ and $ev_1$ in $etr$ might be different events.

The stream extraction algorithm is designed to separate the streams from a sequence of function call events. In this algorithm, a stream is identified by the endpoint handle output by channel open function calls. Then all other events will be added to this stream. According to the channel open mechanisms discussed in Section 4.4, the identifier of the channel and the handle of the endpoint can be retrieved from the channel open function call events.

The input of this algorithm is the sequence of events $etr = (ev_1, ev_2, ..., ev_n)$ from the function call event reconstruction algorithm. Since the events in $etr$ are reconstructed in sequence of the instructions which are ordered by the time of occurrence, the events are implicitly sorted by time of occurrence.

The outputs of the stream extraction algorithms are a set of streams of size $p$, which can be defined as:

$str = (s_1, s_2, ..., s_p)$

According to the channel open mechanisms, two different algorithms are designed, one is for Name pipe and Message Queue, while the other is for TCP and UDP.

### Stream Extraction Algorithm for Named Pipe and Message Queue

This algorithm is designed for the extraction of the streams for Named Pipe and Message Queue. Since for each endpoint of the communication, only one channel open function call is needed to identify the endpoint, it is simple to identify the stream once the channel open function call event

that create the endpoint handle is found.

The same handle may be reused by other endpoint. However, reuse of the handle would not happen before this stream is closed by the channel close function call event. Therefore, before the detection of the channel close function call, if a new channel open function call with the same returned handle is detected, the second channel open is treated as an error. The error handling is not discussed in this algorithm. This algorithm recognizes this error by having $tempstreams$ to keep track of the streams that are still open. Once the stream is closed, this stream will be removed from $tempstreams$. The time complexity of this algorithm is $O(N)$ , $N$ is the number of events in the trace.

**Algorithm 2: Stream Extraction Algorithm for Named Pipe and Message Queue**

```
   /* etr is a sequence of function call events output by Algorithm 1; str is a set of streams
      corresponding to a set of endpoints                                                    */
   Input: etr
   Output: str
1  str ← ∅
   /* a temperate stream set for all open streams                                            */
2  tempstreams ← ∅
3  for ev ∈ etr do
4      if ev.type = open then
5          h ←the handle in ev.outparams
6          if tempstreams[h] not exist then
7              tempstreams[h] ← a new s
8              tempstreams[h].handle ← h
9              tempstreams[h].channelId ← the channel identifier from ev.inparams
10             tempstreams[h].so.append(ev)
11     else if ev.type = send then
12         h ←the handle in ev.inparams
13         if tempstreams[h] exist then
14             tempstreams[h].ss.append(ev)
15     else if ev.type = receive then
16         h ← the handle in ev.inparams
17         if tempstreams[h] exist then
18             tempstreams[h].sr.append(ev)
19     else if ev.type = close then
20         h ← the handle in ev.inparams
21         if tempstreams[h] exist then
22             tempstreams[h].sc.append(ev)
23             str.append(tempstreams[h])
24             remove tempstreams[h] from tempstreams
25     else
26         unknown event type error
27 return str
```

**Stream Extraction Algorithm for TCP and UDP**

This algorithm is designed for extracting the streams for TCP and UDP. In the channel open stage, socket handles are created by function calls of $socket$ in both client and server. In the server side, this created socket is only used for listening to the client's connection. The listening is accomplished by calling the function $accept$. One of the input parameters of the $accept$ function call is the listening socket handle, and the output of it is a new data transmission socket handle.

In this algorithm, each created socket will be identified as a stream. So that the two socket handles in the server side are considered to be two handles for two streams, the stream identified by the listening handle is called the parent stream and the one identified by the data transmission handle is called the child stream. The events in the parent stream contain the information needed for stream matching algorithm for the child stream later, so the child stream will inherit all the events from its parent.

Same as the algorithm for Named pipe and Message Queue, the reused of a handle can only happen after a stream identified by this handle is closed. Otherwise the handle reuse will be treated as an error. The error handling is not discussed in this algorithm. A set $tempstreams$ is also used in this algorithm to keep check of the open streams.

The time complexity of this algorithm is also $O(N)$, $N$ is the number of events in the trace.

## Algorithm 3: Stream Extraction Algorithm for TCP and UDP

```
/* etr is a sequence of function call events output by Algorithm 1; str is a sequence of
   streams corresponding                                                              */
Input: etr
Output: str
str ← ∅;
/* a temperate stream set for all open streams                                        */
tempstreams ← ∅;
for ev ∈ etr do
    if ev.funN = socket then
        h ← the handle in ev.outparams;
        if tempstreams[h] not exist then
            tempstreams[h] ← a new s                          // new a stream;
            tempstreams[h].handle ← h;
            tempstreams[h].so.append(ev);
    else if ev.funN = bind or ev.funN = connect then
        h ← the handle in ev.inparams;
        if tempstreams[h] exist then
            tempstreams[h].channelId ← address and port parameter in ev.inparams;
            tempstreams[h].so.append(ev);
    else if ev.funN = accept then
        h ← the handle in ev.inparams;                  // the handle of parent stream;
        hc ← the handle in ev.outparams;                 // the handle of child stream;
        if tempstreams[h] exist then
            if tempstreams[hc] not exist then
                tempstreams[hc] ← a new s;               // a new stream for the child;
            tempstreams[hc].handle ← hc;
            tempstreams[hc].channelId ← tempstreams[h].channelId;
            tempstreams[hc].so.append(tempstreams[h]);          // append parent's events;
            tempstreams[hc].so.append(ev);                  // append the current event;
    else if ev.type = send then
        h ← the handle in ev.inparams;
        if tempstreams[h] exist then
            tempstreams[h].ss.append(ev);
    else if ev.type = receive then
        h ← the handle in ev.inparams;
        if tempstreams[h] exist then
            tempstreams[h].sr.append(ev);
    else if ev.type = close then
        h ← the handle identifier from ev.paras;
        if tempstreams[h] exist then
            tempstreams[h].sc.append(ev);
            str.append(tempstreams[h]);
            remove tempstreams[h] from tempstreams;
    else
        unknown event type or name error;
return str;
```

## 4.6 Stream Matching Algorithm

The function event extraction algorithm and the stream extraction algorithms work on a single execution trace. As defined before, a communication has two endpoints and each endpoint corresponds to a stream. To identify a communication from the dual_trace, the two streams of that communication need to be found.

The stream matching algorithm iterates over all the streams extracted from both traces of a dual_trace and tries to match one stream of a trace to a stream of the other trace using the channel identifier held by each stream.

The channel identifiers held by the streams are retrieved in the stream extraction algorithm and are different for different communication methods. For TCP and UDP, channel identifier is the server's address and port. For Named Pipe, the channel identifier is the file name, while for Message Queue, the channel identifier is the queue name.

The inputs of this algorithm are two sequence of streams $str_0$ and $str_1$ which are output by the stream extraction algorithm. The output of this algorithm is a sequence of the preliminary communications $cs$ of two matched streams. Each matched item in it is a triple $< channelId, s_0, s_1 >$, where $channelId$ is the identifier of the channel, while $s_0$ and $s_0$ are the streams from $trace_0$ and $trace_1$ that correspond to the communication performed by each program on that channel. The time complexity of this algorithm is $O(N * M)$, $N$ and $M$ are the number of streams in both traces.

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

```
/* str_0 and str_1 are two sequences of streams from trace_0 and trace_1.  cs is a sequence of
   preliminary communications                                                              */
Input: str_0, str_1
Output: cs
1 cs ← ∅
2 for s_0 ∈ str_0 do
3     for s_1 ∈ str_1 do
4         if s_0.channelId = s_1.channelId then
5             Create a new communication c ←< channelId, s_0, s_1 >
6             cs.append(c)
7 return cs
```

This matching algorithm is not fully reliable. There are two situations in which the matching will fail. Take Named Pipe for example, the Named Pipe server is connected by two clients(client1 and client2) using the same file. The server trace and the client1 trace are analyzed as a dual_trace, while the server trace and the client2 trace are analyzed as the other dual_trace. In the server trace, there are two streams found. In each client trace, there is one stream found. For the dual_trace of the server and client1, there will be two possible identified communications, one is the real

communication for server and client1 while the other is an error which actually is for server and client2. The stream in client1's trace will be matched by the two streams in the server's trace.

The second situation is the same channel is reused by the different endpoints in the same programs. For example, the Named Pipe server and client finished the first communication and then closed the channel. After a while they re-open the same file again for another communication. The matching is based on the identifiers. So that in this case, there will be four matching.

Similar situations can also happen in Message Queue, TCP and UDP communication methods.

The data stream verification algorithm discussed in next section can reduce these errors.

## 4.7 Data Stream Verification Algorithm

In the last section, I presented the stream matching algorithm and described the situations in which the matching can go wrong. In this section, I present the algorithms that designed to verify if the data in the two streams of a preliminary identified communication satisfies the communication preservation properties of the communication model in Chapter 3.

The data transfer characteristics divide the communications into reliable and unreliable categories. Named Pipe and TCP fall in the reliable category while Message Queue and UDP fall in the unreliable one. The properties of the models consists of content preservation and timing preservation. So that the verification should cover both preservation properties:

- verify the content preservation of the data in the matched streams.
- verify the timing preservation of the data in the matched streams.

To verify the timing preservation, the relative time of the events in both streams is needed. Unfortunately, we can only determine the relative time with in a stream but not crossing two streams. So that it's unfeasible to verify the timing preservation property for neither reliable nor unreliable communications. The verification algorithms discussed in this section will only cover the content preservation property.

The inputs of the data stream verification algorithms are two preliminary matched streams $s_0$ and $s_1$. The output is the a boolean indicating if the streams satisfy the content preservation. All communications that don't satisfy the content preservation should be excluded as identified communications.

For each communication method the verification of the corresponding preservation is applied, That is, for Named Pipe and TCP, the reliable communication preservation need to be verified and for Message Queue and UDP, the unreliable communication preservation need to be verified. The

following sub sections present the versification algorithms for these four communication methods. In each sub section, I discuss the data transfer properties and scenarios of the communication method and then present the developed verification algorithm.

### 4.7.1 Data Stream Verification Algorithm for Named Pipe

Named Pipe provides FIFO communication mechanism for inter-process communication. It can be a one-way or a duplex pipe. [**?**]

The basic data transfer characteristics of Named Pipe are:

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

Based on these characteristics, the data transfer scenarios of Named pipe can be exemplified in Figure 4.6.

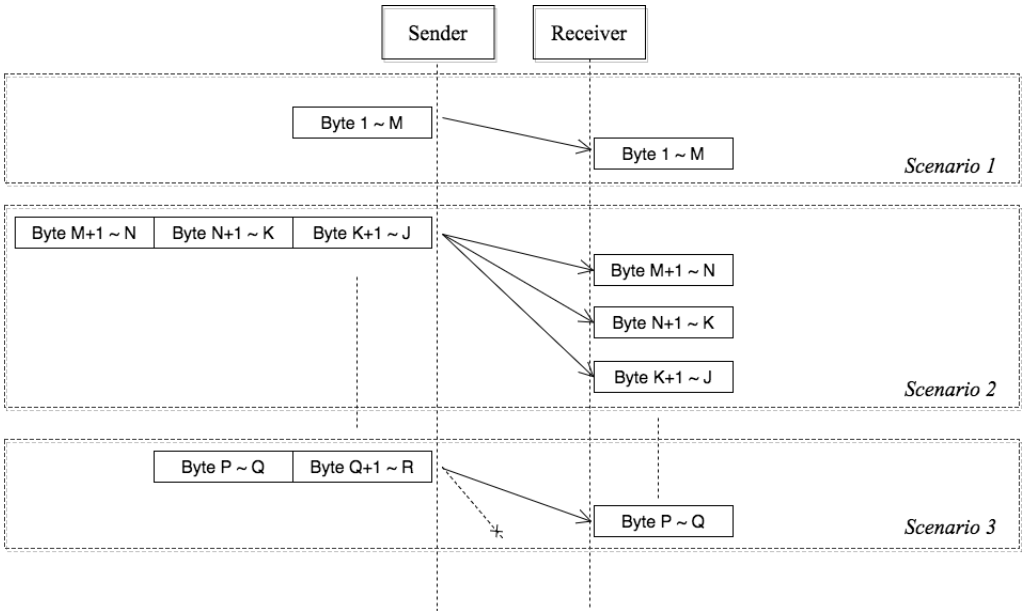



Figure 4.6: Data Transfer Scenarios for Named Pipe

The content preservation verification is trivial. It compares the concatenation of the packet content of the sent events in a stream to the concatenation of the packet content of the receive events in the other stream, which is presented in Algorithm 5. Since the concatenation needs to inspect the events in the streams, the time complexity of this algorithm is $O(N)$, $N$ is the total number of data transfer events in the two streams.

**Algorithm 5: Data Stream Verification of Named Pipe**

/* $s_0$ and $s_1$ are two matched streams from $trace_0$ and $trace_1$. The output boolean $satisfied$ is true if the matched stream satisfy the content preservation of a communication. */

**Input:** $s_0, s_1$

**return** $satisfied$

$send_0 \leftarrow$ concatenation of the payload of send function call events in $s_0.ss$;

$send_1 \leftarrow$ concatenation of the payload of send function call events in $s_1.ss$;

$receive_0 \leftarrow$ concatenation of the payload of receive function call events in $s_0.sr$;

$receive_1 \leftarrow$ concatenation of the payload of receive function call events in $s_1.sr$;

**return** $receive_1$ *is prefix of* $send_0$ *AND* $receive_0$ *is prefix of* $send_1$

### 4.7.2 Data Stream Verification Algorithm for TCP

TCP is the most basic 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 acknowledgement sequence this endpoint is expecting from the other endpoint(if ACK is set).

The basic data transfer characteristics of TCP are:

- Bytes received in order
- No data lost (lost data will be re-transmitted)
- No data duplication
- Bytes sent in packet and received in packet, no re-segmentation

Based on these characteristics, the data transfer scenarios of TCP can be exemplified in Figure 4.7.

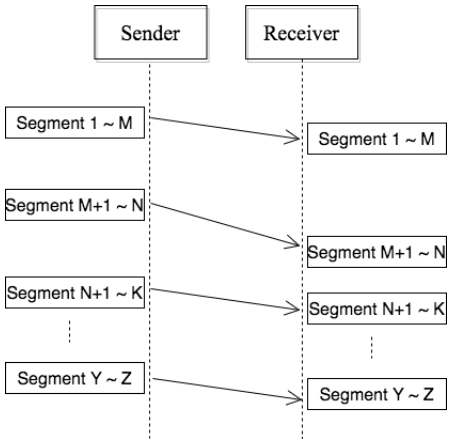



Figure 4.7: Data Transfer Scenarios for TCP

According to the data transfer properties of TCP, all packets sent in one side will be received in the same other in the other side. So that, the verification can be restricted to packet to packet. If every $i-th$ send event in a stream can be matched by the $i-th$ receive event in the other stream for both directions, we can assert that the content preservation is satisfied for the communication consist of these two streams. The verification algorithm of TCP is presented in Algorithm 6. The time complexity of this algorithm is also $O(N)$, $N$ is the number of data transfer events in a stream.

**Algorithm 6: Data Stream Verification of TCP**

```
   /* s_0 and s_1 are two matched streams from trace_0 and trace_1.  The output boolean satisfied is
      true if the matched stream satisfy the content preservation of a communication.      */
   Input: s_0, s_1
1  return satisfied
   /* There is a chance that the trace capturing end before the channel is closed          */
2  for i ∈ 0..min(s_0.ss.size, s_1.sr.size) do
3      if s_0.ss[i].payload! = s_1.sr[i].payload then
4          return False;
5  for i ∈ 0..min(s_1.ss.size, s_0.sr.size) do
6      if s_1.ss[i].payload! = s_0.sr[i].payload then
7          return False;
8  return True;
```

### 4.7.3 Data Stream Verification Algorithm for Message Queue

Message Queue is a communication method to allow applications which are running at different times across heterogeneous networks and systems that may be temporarily offline to communicate with each other. Applications communicate to each other through the queue. Multiple sending applications can send messages to one queue and multiple receiving applications can read messages

from one queue [**?**]. In this work, only the case of one sending application versus one receiving application is considered. A queue can be one-way or duplex.

The basic data transfer characteristics of Message Queue are:

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

Based on these characteristics, the data transfer scenarios of Message Queue can be exemplified in Figure 4.8.

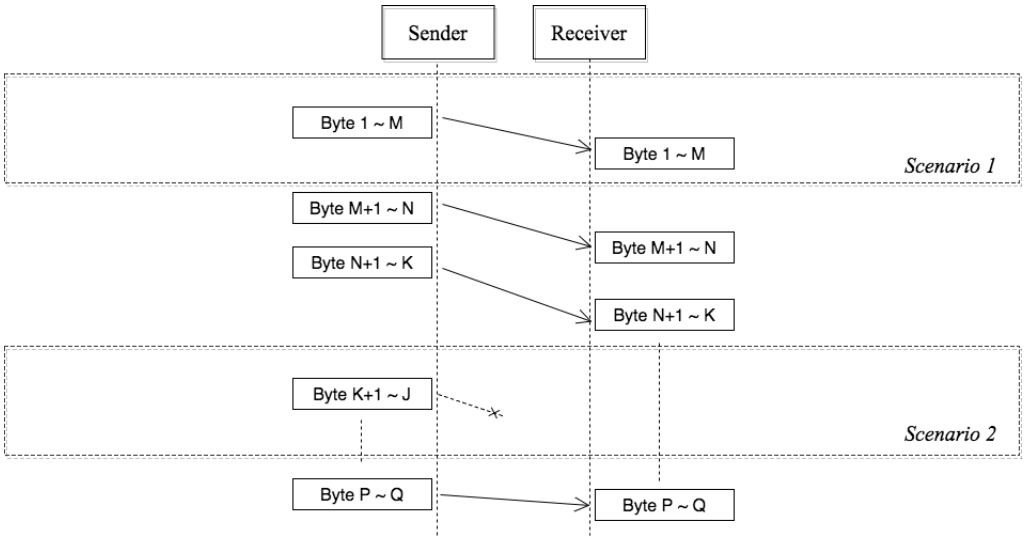



Figure 4.8: Data Transfer Scenarios for Message Queue

To verify the content preservation of the unreliable communication, for each received packet, Algorithm 7 tries to find the match sent packet in the other stream. If any of the received packets can not be matched, the content preservation is not satisfied. Since the sent packets are received in order, the searching for each received packet will start from the next index of the last matched sent packet.

The time complexity of this algorithm is $O(N^2 + M^2)$, $N$ and $M$ are the numbers of data sent events of the two streams.

**Algorithm 7: Data Stream Verification of Message Queue**

```
/* s_0 and s_1 are two matched streams from trace_0 and trace_1. The output boolean satisfied is
   true if the matched stream satisfy the content preservation of a communication.      */
Input: s_0, s_1
return satisfied
if s_0.ss.size < s_1.sr.size Or s_1.ss.size < s_0.sr.size then
    return False;
lastMatchIndex = 0;
for i ∈ 0..s_1.sr.size do
    tempIndex = lastMatchIndex;
    for j ∈ lastMatchIndex + 1..s_0.ss.size do
        if s_0.ss[j].payload = s_1[i].sr.payload then
            lastMatchIndex ← j;
            break the inner For loop;
    /* This received packet can not be matched by any sent packet                          */
    if tempIndex = lastMatchIndex then
        return False;
lastMatchIndex = 0;
for i ∈ 0..s_0.sr.size do
    tempIndex = lastMatchIndex;
    for j ∈ lastMatchIndex + 1..sends_1.size do
        if s_1.ss[j].payload = s_0[i].sr.payload then
            lastMatchIndex ← j;
            break the inner For loop;
    if tempIndex = lastMatchIndex then
        return False;
return True;
```

### 4.7.4 Data Stream Verification Algorithm for 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 be lost
- Packets can be duplicated
- Packets can arrive receiver out of order

Based on these characteristics, the data transfer scenarios of UDP can be exemplified in Figure 4.9.

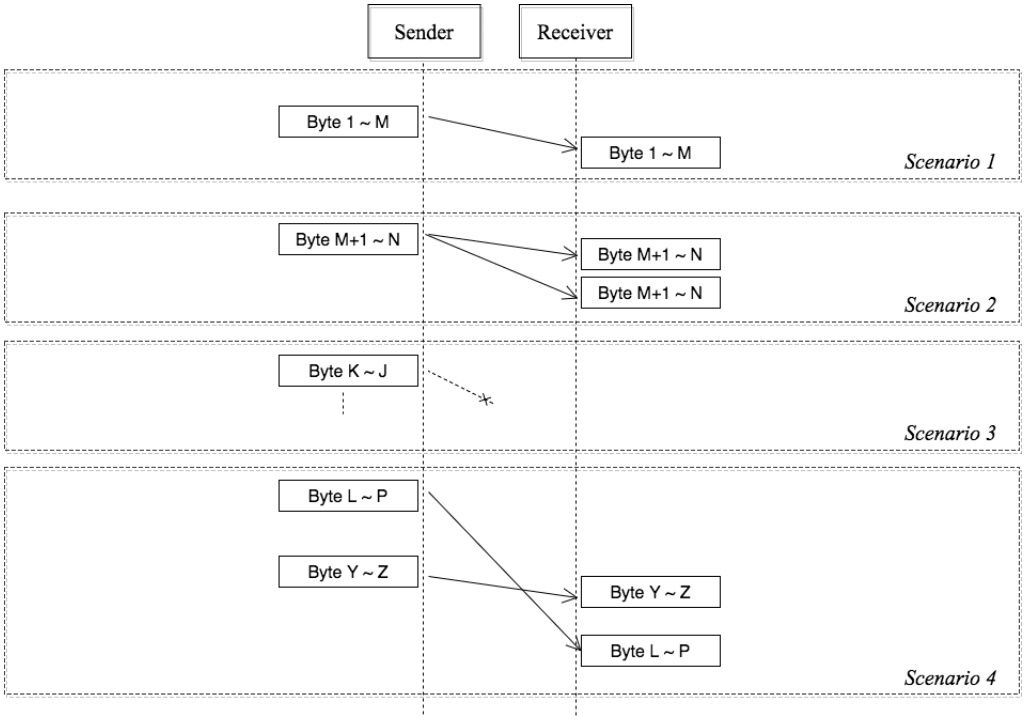



Figure 4.9: Data Transfer Scenarios for UDP

Similar to Message Queue, Algorithm 8 tries to match each received packet in one stream to the corresponding sent packet in the other stream. If any of the received packets can not be matched, the content preservation is not satisfied. However, due to the potential reordering of packet, the matching is not restricted to the packet index(e.g the $i-th$ received packet in one stream can be matched to the $j-th$ packet in the other stream, $i \neq j$). But the matched sent packet will be excluded from the following matching, which means each sent packet can only match to one received packet.

The time complexity of this algorithm is $O(N^2 + M^2)$, $N$ and $M$ are the numbers of data sent transfer events in the two streams.

**Algorithm 8: Transmitted Verification of UDP**

```
/* s_0 and s_1 are two matched streams from trace_0 and trace_1. The output boolean satisfied is
   true if the matched stream satisfy the content preservation of a communication.      */
Input: s_0, s_1
return satisfied
if s_0.ss.size < s_1.sr.size Or s_1.ss.size < s_0.sr.size then
    return False;
/* For each received packet in stream s_1 try to find the sent packet in stream s_0. s_1.sr is
   the sequence of received packet of stream s_1 while s_0.ss is the sequence of sent packet
   of stream s_0                                                                        */
for i ∈ 0..s_1.sr.size do
    matchFlag = False;
    for j ∈ 0..s_0.ss.size do
        if s_0[j].ss.payload = s_1[i].sr.payload then
            matchFlag = True;
            delete the packet from sends_0 break the inner For loop;
    /* This received packet can not be matched by any sent packet                       */
    if matchFlag = False then
        return False;
/* For each received packet in stream s_0 try to find the sent packet in stream s_1. s_0.sr is
   the sequence of received packet of stream s_0 while s_1.ss is the sequence of sent packet
   of stream s_1                                                                        */
for i ∈ 0..s_0.sr.size do
    matchFlag = False;
    for j ∈ 0..s_1.ss.size do
        if s_1[j].ss.payload = s_0[i].sr.payload then
            matchFlag = True;
            delete the packet from s_0.ss break the inner For loop;
    if matchFlag = False then
        return False;
return True;
```

### 4.7.5 Limitation of the Data Verification

The verification discussed in this chapter has two major limitations:

- The timing preservation of the communication is not verified.
- Some mismatching can not be excluded even with the content verification

The reason that the timing preservation can not be verify is lacking of the timing information across the two traces.

For the second limitation, the main reason is that the data transmitted in two communications cloud be identical or very similar.

In Section 4.6, I described how one stream in one side of the dual_trace can be matched by two or more streams on the other side happens. In order to reduce this type of error, I developed the data stream verification strategy and algorithms. However, in some cases, the transferred data of two communications can be identical or very similar. So that even with content verification, the mismatched streams still can not to excluded.

Figure 4.10 exemplifies this situation in a reliable communication analysis. Assuming that $s_{11}$, $s_{21}$ and $s_{22}$ have the same channel identifier, they will be matched as two communications( one consists of $s_{11}$ and $s_{21}$ while the other one consists of $s_{11}$ and $s_{22}$). Furthermore, $s_{21}$ and $s_{22}$ send and receive the exact data. So that, both of the communications are considered to satisfy the content preservation property. In this case, the algorithms will not be able to determine if $s_{11}$ communicated with $s_{21}$ or $s_{22}$. There is no way from the analysis point of view to distinguish the actual communication. It's more reasonable to preserve all of them in the output and present them for the users to make the decision. The users might be able to distinguish them and find the valid one.

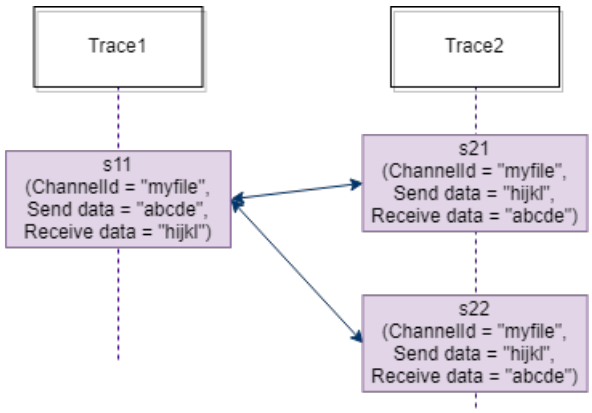



Figure 4.10: Ineffective Stream Matching Scenario

# Chapter 5

# Dual_trace Communication Analysis Prototype On Atlantis

In this chapter, I present the design of the prototype of communication analysis from the dual_trace. This prototype is implemented on Atlantis. Atlantis as an assembly execution trace analysis environment. It provides many features that benefit the communication analysis of dual_trace.

This prototype consists of four components: 1) declaring the function descriptors of the communication methods. 2) a view that can display both traces in the dual_trace in parallel. 3)implementation of the communication analysis algorithms for stream extraction and communication identification. 4) a view for presenting the extracted streams and the identified communications from the dual_trace.

## 5.1 Use Case

In this section, the use cases are presented to depict how to use the developed components and the existing features of Atlantis to perform the communication analysis.

To analyze a dual_trace, the user need to perform the below operations in sequence:

- Open two traces in the parallel view (as part of this prototype)
- Import the dynamic linked library files for each trace (as part of existing functionality of Atlantis)
- Perform the stream extraction or communication identification operations(as part of this prototype)

- Inspect the operation results by navigating the analysis result from the communication view (as part of this prototype).

Two use cases of this prototype are designed. The use case shown in Table 5.1 is for stream extraction and the use case shown in Table 5.2 is for communication identification.

Table 5.1: Use Case 1: Extract Streams from the Dual_trace

| | |
|---|---|
| **Name** | Analysis streams of a communication method from the Dual_trace |
| **Description** | A user captured two assembly execution traces of two interacting programs and needs to analysis them by extracting all communication streams of each of the traces and inspecting the extraction results |
| **Actor** | A Software Security Engineer |
| **Precondition** | The user has two assembly execution traces and the .dll files of the systems where the programs of the captured traces were running |
| **Main Course** | 1. The user declares the function descriptors for the communication methods of interest in a Json format setting file<br>2. The user opens one of the trace in Atlantis<br>3. The user opens the other trace as the dual_trace of the first one<br>4. The two opened traces are presented in the parallel view<br>5. The user loads the related .dll files for both opened trace<br>6. The user selects the operation "Stream Extraction" in the "Dual_Trace Tool" menu.<br>7. Atlantis prompts a dialog window giving the user all the communication methods in the function descriptor setting file as options<br>8. The user selects the communication methods which they want to analyze and click the "OK" bottom<br>9. Atlantis extracts the streams for both traces and list the result in "Communication view"<br>10. The user expands the result in the "Communication view"<br>11. The user selects one function call event in a stream and double click the entry<br>12. Atlantis shows the corresponding instruction line in the trace and synchronizes all other views |

Table 5.2: Use Case 2: Identify Communications from the Dual_trace

| | |
|---|---|
| **Name** | Identify communications of a communication method from the Dual_trace |
| **Description** | A user captured two assembly execution traces of two interacting program and need to analysis them by identify all communications of the dual_trace and inspecting the extraction results |
| **Actor** | A Software Security Engineer |
| **Precondition** | The user has two assembly execution traces and the .dll files of the systems where the programs of the captured traces were running |
| **Main Course** | 1. The user declares the function descriptors for the communication methods of interest in a Json format setting file<br>2. The user opens one of the trace in Atlantis<br>3. The user opens the other trace as the dual_trace of the first one<br>4. The two opened traces are presented in the parallel view<br>5. The user loads the related .dll files for both opened trace<br>6. The user selects the operation "Communication Identification" in the "Dual_Trace Tool" menu<br>7. Atlantis prompts a dialog window giving the user all the communication methods in the function descriptor setting file as options<br>8. The user selects the communication methods which they want to analyze and click the "OK" bottom<br>9. Atlantis identifies the communications of the dual_trace and list the result in "Communication view"<br>10. The user expands the result in the "Communication view"<br>11. The user selects one function call event in a stream and double click the entry<br>12. Atlantis shows the corresponding instruction line in the trace and synchronize all other views |

## 5.2 Declaring of the Function Descriptors

In Section 4.2, I describe how to define a function descriptor for each communication method. Each function description consist with four elements:

$fdesc = \{name, type, inparamdesc, outparamdesc\}$

$name$ is the function name, $type$ can be $open$, $close$, $send$ and $receive$, $inparamdesc$ and $outparamdesc$ are the descriptions for the input and output parameters of interest(we might not care for all parameters). The communication analysis approach depicted in Chapter 4 can identify the communications described by the function descriptor from the dual_trace.

However, the function descriptor for a communication method can be different depending on the implementation of the communication method in a program. Rather than hard coding the function descriptors for the communication methods, the prototype loads the function descriptors from a configuration file. A default template is given for reference. This template is generated by Atlantis when it is launched and stored in the .tmp folder of the trace analysis project. The users can modify this template as to the communication methods of interest. The default template example can be found in Appendix B.

The function descriptors in this configuration file will be the input for the stream extraction and communication identification features. When the user uses these two features, the list of the communication methods provided in the function descriptor configuration file will be presented to them. They can select one or more communication methods to be analyzed.

In the following subsections, function descriptor examples are presented for reference. Other function descriptors can be created by following the same method as developing the function descriptor examples.

### 5.2.1 Communication Methods' Implementation in Windows

Learning the implementation of a communication is necessary to obtain the function descriptor of the communication method. In this section, I present the results of analyzing the implementation of these four communication methods: Named Pipe, Message Queue, TCP and UDP in Windows. In the analysis, I reviewed the Windows APIs of the communication methods and their example code. By doing so, I obtained the function descriptors of these methods.

The Windows API set is very complex. Moreover, multiple solutions are provided to fulfil a communication method. It is unfeasible within the scope of this thesis to enumerate all solutions for each communication method. I only investigated the most basic usage provided in Windows documentation. For each communication method, a function descriptor with a list of system function descriptions is provided for reference. The functions in the descriptors are supported in most Windows operating systems, such as Windows 8, Window 7. The provided function descriptor of a communication method should only be considered as an example for that communication method. With the understanding of this, it should be fairly easy to obtain the function descriptors for other solutions of that communication method or other communication methods.

Note that, the instances of the descriptors only demonstrate Windows C++ APIs. But the idea of the function descriptor is generalizable to other operating systems with the effort of understanding the APIs of those operating systems.

### Windows Calling Convention

Ror this research, it is important to know the Windows calling convention. The communication analysis from a dual_trace in assembly level relies not only on the system function names but also the key parameter values and return values. In the assembly level execution traces, the parameter and return values are captured in the memory changes and register changes of the instructions but without any explicit information indicating which registers or memory addresses are holding these parameters. The calling convention tells us where the parameters are stored. So that, we can find them in the memory state while emulating the execution of the trace. Each operating system has their own calling convention for different programming languages. I used dual_traces of Microsoft* x64 programs for case study in this research. The Microsoft* x64 calling convention is listed in Appendix A for reference.

### Named Pipes

In Windows, a named pipe is a communication method between one server and one or more clients. The pipe has a name and can be one-way or duplex. Both the server and clients can read or write into the pipe[**?**]. In this work, I only consider one server versus one client communication (one server to multiple clients scenario can always be divided into multiple "one server and one 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" and 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 operations. The function descriptors for both synchronous mode and asynchronous mode are provided. The create channel functions for both modes are the same while the mode is indicated by an input parameter. The functions for send and receive message are also the same for both cases. However, the operations 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 finished, the output message will be stored in the overlap structure whose memory address saved in

the function's output parameter OverlapStruct. Table 5.3 is the function descriptor for synchronous mode while Table 5.4 is the function descriptor asynchronous mode of Named pipe.

Table 5.3: Function Descriptor for Synchronous Named Pipe

| **Name** | **Type** | **Input Parameters Description** | | | **Output Parameters Description** | | |
|---|---|---|---|---|---|---|---|
| | | **Name** | **Register** | **Addr/Val** | **Name** | **Register** | **Addr/Val** |
| CreateNamedPipe | open | FileName | RCX | Addr | Handle | RAX | Val |
| CreateFile | open | FileName | RCX | Addr | Handle | RAX | Val |
| WriteFile | send | Handle | RCX | Val | Length | R9 | Val |
| | | SendBuf | RDX | Addr | RetVal | RAX | Val |
| ReadFile | receive | Handle | RCX | Val | Length | R9 | Val |
| | | RecvBuf | RDX | Addr | RetVal | RAX | Val |
| CloseHandle | close | Handle | RCX | Val | RetVal | RAX | Val |
| DisconnectNamedPipe | close | Handle | RCX | Val | RetVal | RAX | Val |

Table 5.4: Function Descriptor for Asynchronous Named Pipe

| **Name** | **Type** | **Input Parameters Description** | | | **Output Parameters Description** | | |
|---|---|---|---|---|---|---|---|
| | | **Name** | **Register** | **Addr/Val** | **Name** | **Register** | **Addr/Val** |
| CreateNamedPipe | open | FileName | RCX | Addr | Handle | RAX | Val |
| CreateFile | open | FileName | RCX | Addr | Handle | RAX | Val |
| WriteFile | send | Handle | RCX | Val | Length | R9 | Val |
| | | SendBuf | RDX | Addr | RetVal | RAX | Val |
| ReadFile | receive | Handle | RCX | Val | Length | R9 | Val |
| | | RecvBuf | RDX | Addr | RetVal | RAX | Val |
| GetOverlappedResult | receive | Handle | RCX | Val | OverlapStruct | RDX | Addr |
| | | | | | RetVal | RAX | Val |
| CloseHandle | close | Handle | RCX | Val | RetVal | RAX | Val |
| DisconnectNamedPipe | close | Handle | RCX | Val | RetVal | RAX | Val |

### Message Queue

Similar to Named Pipe, the Message Queue's implementation in Windows also has two modes, synchronous and asynchronous. The asynchronous mode is also further divided into two operations: one with callback function and the other without. With the callback function, the callback

function would be called when the send or receive operations finish. Without a callback function, the general function *MQGetOverlappedResult* should be called by the endpoints to check if the message sending or receiving operation finish, the parameter in RCX of this function call is a structure consist of the handle as an input parameter and the overlap structure as an output parameter. Table 5.5 is the function descriptor for synchronous mode while Table 5.6 is the function descriptor for the asynchronous mode without callback. I did not exemplify the case the with callback function, since the specific callback function and its parameters need to be known for developing this the function descriptor.

Table 5.5: Function Descriptor for Synchronous Message Queue

| **Name** | **Type** | **Input Parameter Description** | | | **Output Parameter Description** | | |
|---|---|---|---|---|---|---|---|
| | | **Name** | **Register** | **Addr/Val** | **Name** | **Register** | **Addr/Val** |
| MQOpenQueue | open | QueueName | RCX | Addr | Handle | RAX | Val |
| MQSendMessage | send | Handle | RCX | Val | RetVal | RAX | Val |
| | | MessStruct | RDX | Addr | | | |
| MQReceiveMessage | receive | Handle | RCX | Val | MessStruct | RDX | Addr |
| | | | | | RetVal | RAX | Val |
| MQCloseQueue | close | Handle | RCX | Val | RetVal | RAX | Val |

Table 5.6: Function Descriptor for Asynchronous Message Queue

| **Name** | **Type** | **Input Parameter Description** | | | **Output Parameter Description** | | |
|---|---|---|---|---|---|---|---|
| | | **Name** | **Register** | **Addr/Val** | **Name** | **Register** | **Addr/Val** |
| MQOpenQueue | open | QueueName | RCX | Addr | Handle | RAX | Val |
| MQSendMessage | send | Handle | RCX | Val | RetVal | RAX | Val |
| | | MessStruct | RDX | Addr | | | |
| MQReceiveMessage | receive | Handle | RCX | Val | MessStruct | RDX | Addr |
| | | | | | RetVal | RAX | Val |
| MQGetOverlapResult | receive | Handle | RCX | Addr | Overlapstr | RCX | Addr |
| | | | | | RetVal | RAX | Val |
| MQCloseQueue | close | Handle | RCX | Val | RetVal | RAX | Val |

**TCP and UDP**

In Windows programming, these two methods share the same set of APIs regardless of whether the input parameter values and operation behavior are different. In the Windows socket solution, one of the two endpoints is the server while the other one is the client. Table 5.7 is the function descriptor for UDP or TCP communication.

Table 5.7: Function Descriptor for for TCP and UDP

| **Name** | **Type** | **Input Parameters Description** | | | **Output Parameters Description** | | |
|---|---|---|---|---|---|---|---|
| | | **Name** | **Register** | **Addr/Val** | **Name** | **Register** | **Addr/Val** |
| socket | open | | | | Socket | RAX | Val |
| bind | open | Socket | RCX | Val | ServerAddrAndPort | RDX | Addr |
| connect | open | Socket | RCX | Val | ServerAddrAndPort | RDX | Addr |
| accept | open | ListenSocket | RCX | Val | ConnectSocket | RAX | Val |
| send | send | Handle | RCX | Val | RetVal | RAX | Val |
| | | SendBuf | RDX | Addr | | | |
| recv | receive | Handle | RCX | Val | RecvBuf | RDX | Addr |
| | | | | | RetVal | RAX | Val |
| closesocket | close | Handle | RCX | Val | RetVal | RAX | Val |

## 5.3 Parallel Trace View For Dual_Trace

The dual_trace consists of two execution traces which are interacting with each other. I have implemented a view that shows the traces side by side. It's called the parallel trace view. Presenting two traces in the parallel trace view makes the analysis for the user much easier. A parallel trace view implemented in Atlantis can display two execution traces side by side. To open the parallel trace view, the user needs to open one trace as the normal one and the other as the dual_trace of the active (opened) one. A new menu option in the project navigation view of Atlantis was added to open the second trace as the dual_trace of the active one. The implementation of the parallel view takes advantage of the existing SWT of Eclipse plug-in development. The detail of the implementation can be found in Appendix 5.2. Figure 5.1 shows the "Open as Dual_Trace" menu option and Figure 5.2 shows the parallel trace view with two traces displaying side by side.

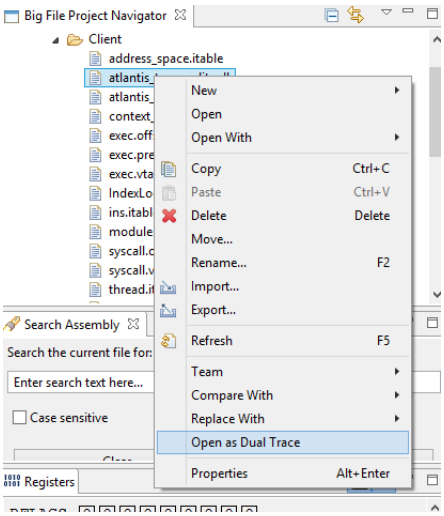



Figure 5.1: Menu Item for opening Dual_trace

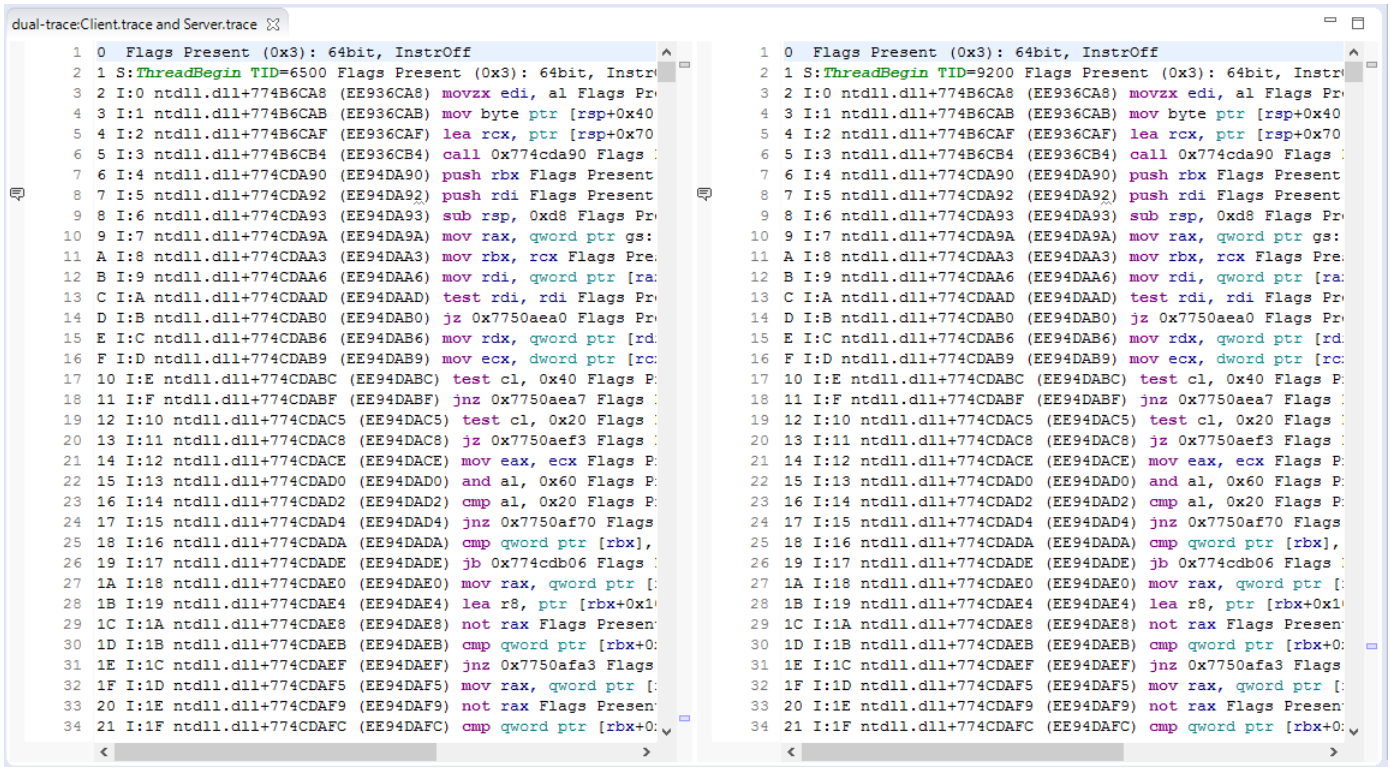



Figure 5.2: Parallel Trace View

## 5.4 Implementation of the Communication Analysis Algorithms

The implementation is divided into two parts. The first part is the stream extraction. It implements the first section of the overall process of the communication analysis as shown in the left side of Figure 5.3 and presents the extracted streams of both traces in the dual_trace to the user. The second part is the communication identification. It implements the whole process of the communication analysis as shown in the right side of Figure 5.3 and presents the identified communications of the dual_trace to the user.

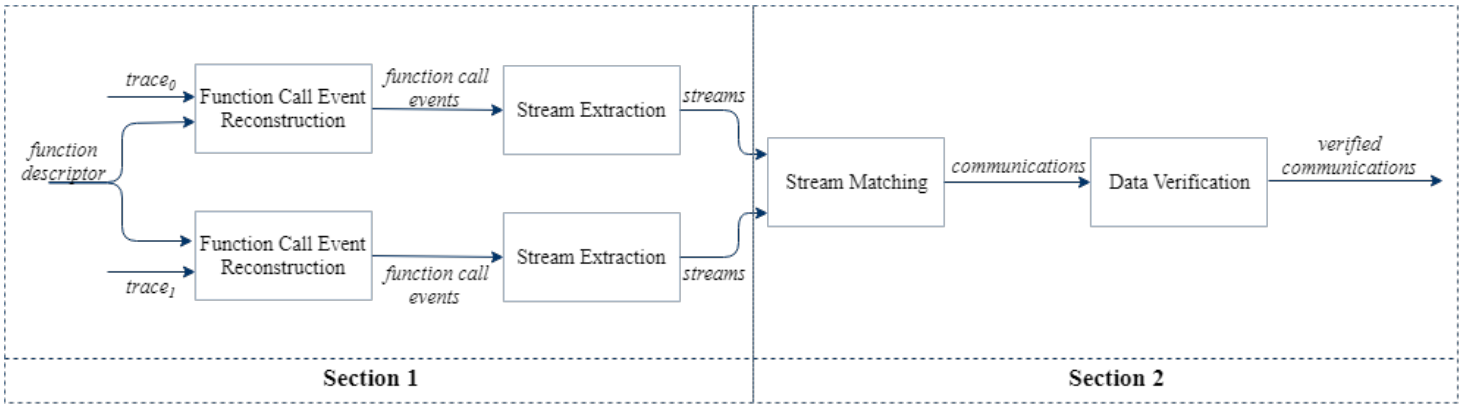



Figure 5.3: Process of the Communication Analysis from a Dual_trace Separated in Two Sections

¡¡¡¡¡¡¡ HEAD The implementation relies on the existing “function inspect” feature of Atlantis. The called functions’ name can be inspected by searching 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 using 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 these two operations.

A new menu “Dual_trace Tool” with three menu options is designed for these two operations. In this menu, “Stream Extraction” is for the operation of stream extraction and “Communication Identification” is for the operation of communication identification. Before performing both of these two operation, “Load Library Exports” has to be run for both traces. Currently, the “Load library export” operation can only load libraries for the trace in the active trace view. So “Load library export” in the menu has to be run twice separately for each trace of the dual_trace. Figure 5.6 shows this new menu in Atlantis. When the user perform “Stream Extraction” or “Communication Identification”, there will be a prompt dialog window as shown in Figure 5.7 which asks the user what communication methods they want to analyze from the dual_trace. This list is provided by the configuration file I mention in Section 5.2. The user can select one or multiple meth-

ods. Atlantis will perform the operations after the user selects and confirms the communication methods. ======= The format of the traces I got from DRDC for this research is slightly different from the formulation of the trace in Section 4.1. In stead of having the function name in $syscallInfo$, the current traces only provide the offset of that function in the corresponding executable file. So that, the trace needs to be processed to comply with the trace formalization. The existing "function inspect" feature of Atlantis can provide this pre-processing functionality. 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. Figure 5.4 is an example of the trace from DRDC. In this example, line 1504 is a system call entry with the system call info as $syscallInfo =< kernal32.dll, 0x10D10 >$. Figure 5.5 is the information decoded by the "function inspect" feature of Atlantis from the executable file kernal32.dll of the system where the trace shown in Figure 5.4 was captured. So that the instruction line 1504 is a system call entry of $CreateFileW$. By loading a .dll file (in this case kernel32.dll) for the trace, Atlantis can translate system call information in the current trace format the one align to the dual_trace formalization.

| Line | Instruction | Memory Changes | Regitster Changes | Execution Type | System Call Info |
|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... |
| $l_{1499}$ | lea rcx, ptr [rip+0x3cfe0] | | ECX F8A0AAA8 | Instruction | |
| $l_{1500}$ | xor r9d, r9d | | | Instruction | |
| $l_{1501}$ | mov edx, 0xc0000000 | | EDX C0000000 | Instruction | |
| $l_{1502}$ | mov dword ptr [rsp+0x20], r8d | 00000000001DF490 00000003 00000000 | RSP 00000000 001DF470 | Instruction | |
| $l_{1503}$ | call qword ptr [rip+0x3a97d] | 00000000001DF470 000007FE F89CDADB | RSP 00000000 001DF468 | Instruction | |
| $l_{1504}$ | mov qword ptr [rsp+0x8], rbx | 00000000001DF470 00000000 00000001 | EBX 00000001 | System Call Entry | kernel32.dll+0x10d10 |
| $l_{1505}$ | mov qword ptr [rsp+0x10], rbp | 00000000001DF478 00000000 00000000 | EBP 00000000 ESP 001DF468 | Instruction | |
| ... | ... | ... | ... | ... | ... |
| $l_{2057}$ | add rsp, 0x20 | | ESP 001DF2A0 | Instruction | |
| $l_{2058}$ | pop rbx | | RBX 00000000 001DF3E8 RSP 00000000 001DF2A8 | Instruction | |
| $l_{2059}$ | ret | | RSP 00000000 001DF2B0 | System Call Exit | kernel32.dll+0x10d10 |
| $l_{2060}$ | mov eax, dword ptr [rsp+0x54] | | EAX 00000000 | Instruction | |
| ... | ... | ... | ... | ... | ... |

Figure 5.4: An example trace from DRDC

| Offset | Name |
|---|---|
| 0x83f0 | CreateFiberEx |
| 0x21b80 | CreateFileA |
| 0xdf80 | CreateFileMappingA |
| 0x65240 | CreateFileMappingNumaA |
| 0x4c9f0 | CreateFileMappingNumaW |
| 0xee90 | CreateFileMappingW |
| 0x75c10 | CreateFileTransactedA |
| 0x75a70 | CreateFileTransactedW |
| 0x10d10 | CreateFileW |
| 0x5bc90 | CreateHardLinkA |
| 0x5bdf0 | CreateHardLinkTransactedA |
| 0x5bd30 | CreateHardLinkTransactedW |
| 0x5b990 | CreateHardLinkW |
| 0x4f80 | CreateIoCompletionPort |
| 0x5c1e0 | CreateJobObjectA |

Figure 5.5: Information from kernal32.dll

A new menu "Dual_trace Tool" with three menu options is designed for these two operations.

In this menu, “Stream Extraction” is for the operation of stream extraction and “Communication Identification” is for the operation of communication identification. Before performing both of these two operation, “Load Library Exports” has to be run for both traces. “Load Library Exports” option in this menu triggers the “function inspect” operation for the active trace. After this operation being run separately for both traces in the dual_trace. The traces are transformed into the proper format for communication analysis. Figure5.6 shows this new menu in Atlantis. When the user perform “Stream Extraction” or “Communication Identification”, there will be a prompt dialog window as shown in Figure 5.7 which asks the user what communication methods they want to analyze from the dual_trace. This list is provided by the configuration file I mentioned in Section 5.2. The user can select one or multiple methods. Atlantis will perform the operations after the user select and confirm the communication methods.

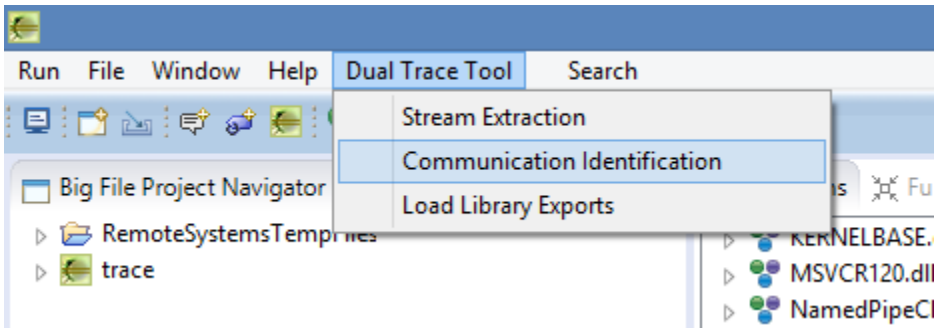



Figure 5.6: Dual_trace Tool Menu

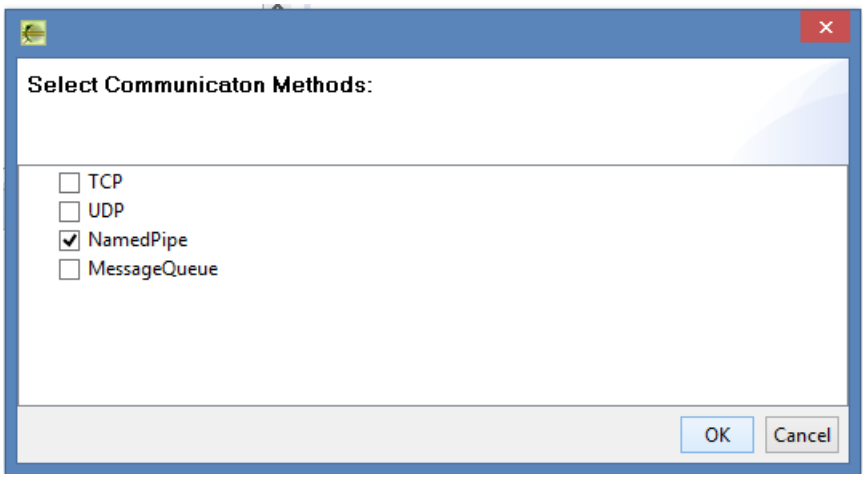



Figure 5.7: Prompt Dialog for Communication Selection

## 5.5 View of Extracted Streams and Identified Communications

A new view named “Communication View” is designed for presenting the result of the extracted streams and the identified communications. Since the user might have selected multiple selection for communication methods of interest, the output result contains all the streams or communications of all selected communication methods and the results are clustered by methods. There are two sub tables in this view, the left one is for presenting the extracted streams while the left one is for presenting identified communications. The reason for putting these two results in the same view is for easy access to and comparison of the data for the users. Figure 5.8 shows this view with both extracted stream results and identified communication results in it. Each time when the user reruns the operations the result in the corresponding table will be refreshed to show only the latest result of that operation. But the other table will not be affected. For example, if the user run the “Stream Extraction” operation first, the extracted streams will be shown on the left table of the view. And then if the user performs the “communication Identification” operation, the identified communications result will be shown on the right table while the left one still holds the last stream extraction result.

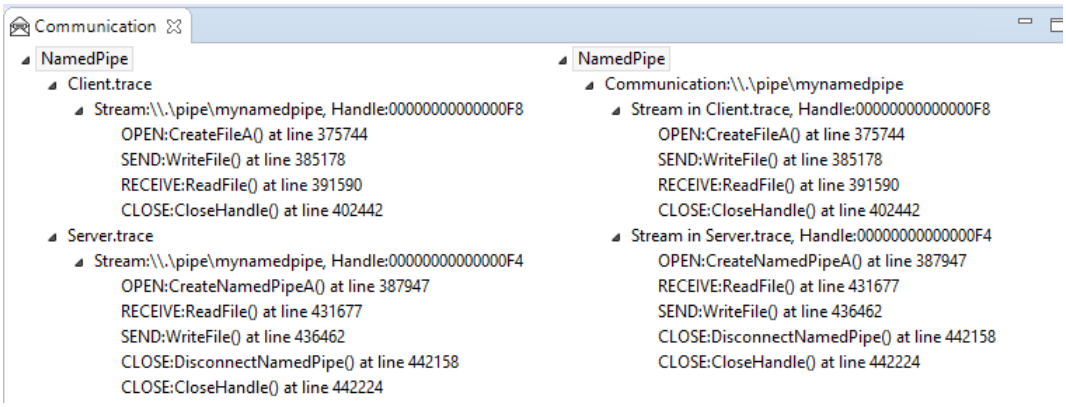



Figure 5.8: Communication View for Result

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

The results presented in the Communication view contains all the function call events. The

memory state at the instruction line of the function begin contains the input parameters and the memory state at the instruction line of the function return contains the output parameters and the return value. In order to provide the user an method to easily access these two instruction lines, 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 clicks" on an entry, it will bring the user to the start line of the function in the corresponding trace view. When the user right clicks on the event entry, a prompted menu with the option "Go To Line of Function End" will show up as shown in Figure 5.9. Clicking on this option will bring the user to the return line of this function in the trace view. All other opened views of Atlantis update immediately with this navigation. By these navigations, the users can easily see the sent and received message of the events.

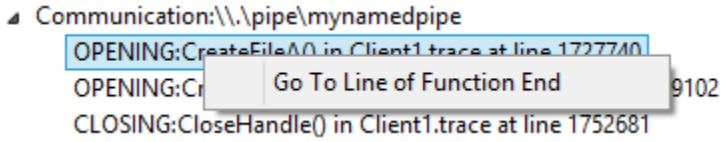



Figure 5.9: Right Click Menu on Event Entry

Moreover, the "remove" option, as shown in Figure 5.10, in the right click menu on the "stream or "communication" entries is provided for the user to remove the selected "stream" or "communication" entry. This provides the users the flexibility to get rid of data that they are not interested about.

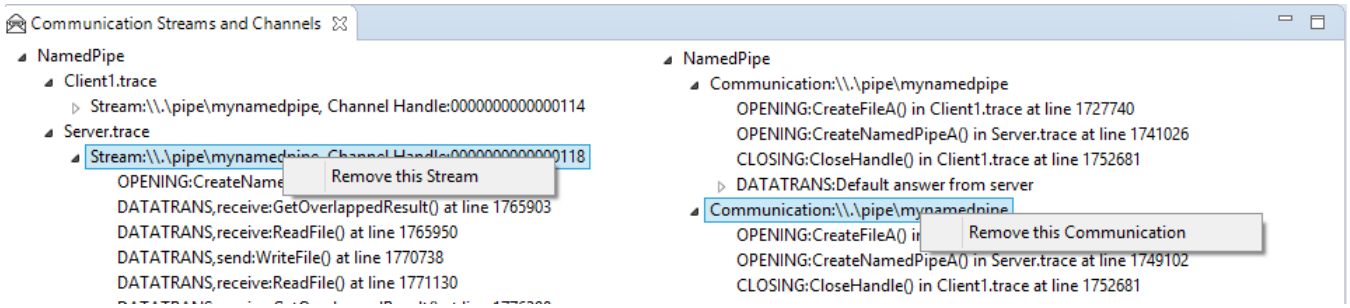



Figure 5.10: Right Click Menu on Event Entry

# Chapter 6

# Proof of Concept

In this chapter, I present two experiments I ran as a proof of concept of the communication analysis of dual_trace.

These experiments were aimed to test the communication model and the communication analysis approach. They also verify the correctness of the algorithms. I used the implemented features on Atlantis to conduct the experiments.

Among these two experiments, the first one was provided directly by our research partner DRDC with their initial requirement, while the second one was designed by me. In both experiments, DRDC conducted the programs execution and captured the traces on their environment while I performed the analysis locally with Atlantis on my desktop with the captured traces. All test programs in these two experiments were written in C++ and the source code can be found in Appendix D.

In both experiments, three major steps are conducted:

1. Execute the test programs and capture the traces for each program (done by DRDC)
2. Perform the "Stream Exaction" and "Communication Identification" operation on the dual_trace
3. Manually verify the result by navigating the events in the "Communication view" and check if they match the sequence diagrams of each experiment

In two following two sections, I will describe the design of the experiments first and then present the result of them with discussion of each one.

## 6.1 Experiment 1

### 6.1.1 Experiment Design

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. Figure 6.1 is the sequence diagram of the interaction between the server and client. This sequence diagram only exemplifies a possible sequence of the events. The actual event sequence can vary depending on the run-time environment.

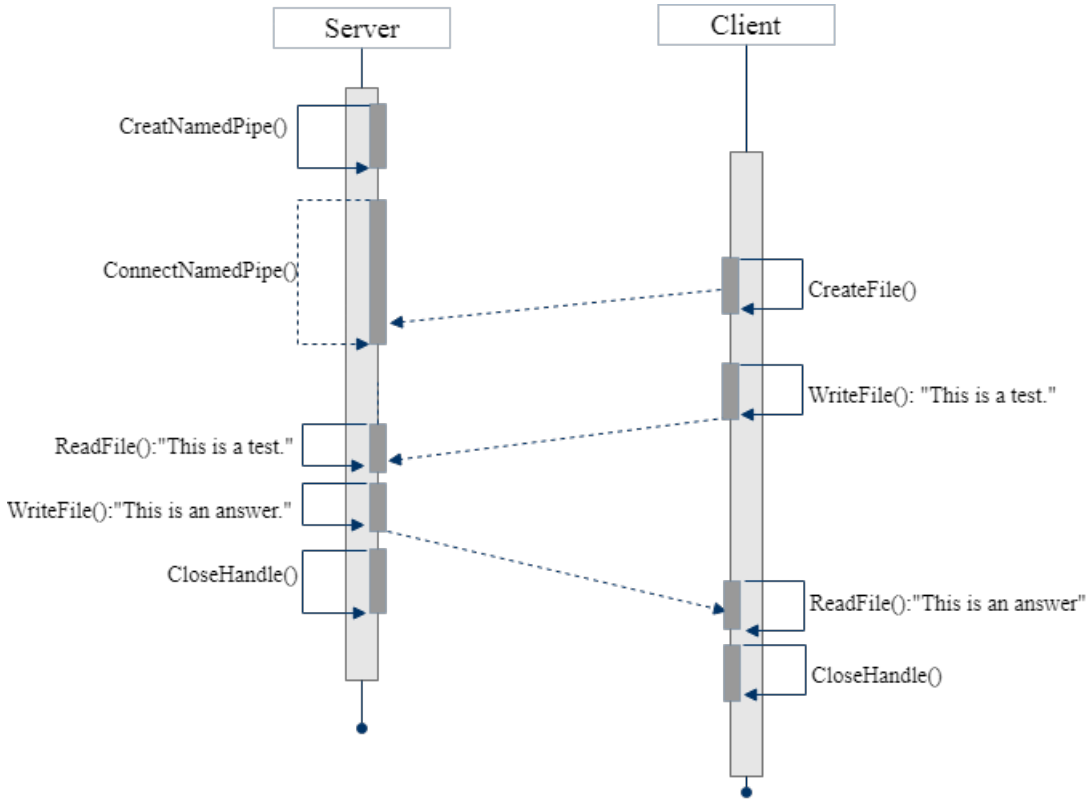



Figure 6.1: Sequence Diagram of Experiment 1

The traces of these programs running and interacting were captured. The two captured traces, $Sever.trace$ and $Client.trace$ were analyzed as a dual_trace and named $dual\_trace\_1$. I performed "Stream identification" and "Communication identification" following the use cases in Table 5.1 and Table 5.2 with $dual\_trace\_1$ and the corresponding kernel32.dll. The function descriptor for the analysis of $dual\_trace\_1$ is shown in Table 6.1.

Table 6.1: Function Descriptor of Named Pipe for Experiment 1

| Name | Type | Input Parameters Description | | | Output Parameters Description | | |
|---|---|---|---|---|---|---|---|
| | | Name | Register | Addr/Val | Name | Register | Addr/Val |
| CreateNamedPipeA | open | FileName | RCX | Addr | Handle | RAX | Val |
| CreateFileA | open | FileName | RCX | Addr | Handle | RAX | Val |
| WriteFile | send | Handle | RCX | Val | Length | R9 | Val |
| | | SendBuf | RDX | Addr | RetVal | RAX | Val |
| ReadFile | receive | Handle | RCX | Val | Length | R9 | Val |
| | | RecvBuf | RDX | Addr | RetVal | RAX | Val |
| CloseHandle | close | Handle | RCX | Val | RetVal | RAX | Val |
| DisconnectNamedPipe | close | Handle | RCX | Val | RetVal | RAX | Val |

### 6.1.2 Dual_trace Analysis Result Walk Through

$Client.trace$ has 412717 instruction lines. Four function call events were reconstructed from this trace as listed in Table 6.2.

Table 6.2: The sequence of function call events of $Client.trace$

| Line | Event |
|---|---|
| 375744 | $funN: CreateFileA, type: open, inparams: \{Handle: 0xF8, FileName: \text{“.\pipe\mynamepipe”}\}, outparams: \{RetVal: 0\}$ |
| 385178 | $funN: WriteFile, type: send, inparams: \{Handle: 0xF8, SendBuf: \text{“This is a test.”}\}, outparams: \{Length: 15\}$ |
| 391590 | $funN: ReadFile, type: receive, inparams: \{Handle: 0xF8\}, outparams: \{RecvBuf: \text{“This is the answer.”}, Length: 18, RetVal: 0\}$ |
| 402442 | $funN: CloseHandle, type: close, inparams: \{Handle: 0xF8\}, outparams: \{RetVal: 0\}$ |

The values of the handle parameter of these four events are $0xF8$. So that a stream identified by the handle $0xF8$ was extracted, which consists of all these four function call events.

$Server.trace$ has 461817 instruction lines. Five function call events were reconstructed from this trace as listed in Table 6.3.

Table 6.3: The sequence of function call events of $Server.trace$

| Line | Event |
|---|---|
| 387947 | $funN: CreateNamedPipeA, type: open, inparams: \{Handle: 0xF4, FileName: \text{“.\pipe\mynamepipe”}\}, outparams: \{RetVal: 0\}$ |
| 431677 | $funN: ReadFile, type: receive, inparams: \{Handle: 0xF4\}, outparams: \{RecvBuf: \text{“This is a test.”}, Length: 15, RetVal: 0\}$ |
| 436462 | $funN: WriteFile, type: send, inparams: \{Handle: 0xF4, SendBuf: \text{“This is the answer.”}\}, outparams: \{Length: 18\}$ |
| 442158 | $funN: DisconnectNamedPipe, type: close, inparams: \{Handle: 0xF4\}, outparams: \{RetVal: 0\}$ |
| 442224 | $funN: CloseHandle, type: close, inparams: \{Handle: 0xF4\}, outparams: \{RetVal: 0\}$ |

The values of the handle parameter of these five events are $0xF4$. So that, a stream identified by the handle $0xF8$ was extracted, which consists of all these five function call events.

The extracted streams were listed in the left table of "Communication view" as shown in Figure 6.2.

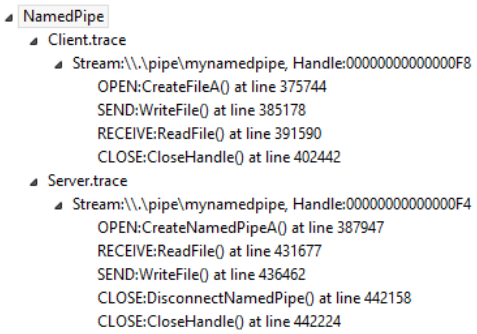



Figure 6.2: Extracted Streams of $dual\_trace\_1$

The value of the $FileName$ parameter of the $CreateFileA$ function call event in $Client.trace$ is "$.\backslash pipe\backslash mynamepipe$" as shown in Table 6.2. Meanwhile, the value of the $FileName$ parameter of the $CreateNamePipeA$ function call event in $Server.trace$ is also "$.\backslash pipe\backslash mynamepipe$" as shown in Table 6.3. According to the algorithm presented in Section 4.6, the file name of a named pipe is treated as the channel identifier which is used to match two streams into a communication. So that. the stream identified by the handle $0xF8$ in $Client.trace$ is matched to the stream identified by the $0xF4$ in $Server.trace$.

There is only one send event and one receive event in both streams, the data verification of these two matched streams is trivial. Both the concatenation of the sent packet(s) in the stream $0xF8$ of $Client.trace$ and the concatenation of the received packet(s) in the stream $0xF4$ of $Server.trace$ are all "$This\ is\ a\ test.$". Both the concatenation of the sent packet(s) in the stream $0xF4$ of $Server.trace$ and the concatenation of the received packet(s) in the stream $0xF8$ of $Client.trace$ are "$This\ is\ the\ answer.$". So these two match stream satisfy the content preservation of the reliable communication. Therefore they are eventually output as a communication by the "Communication Identification" operation and listed in the right table of "Communication view" of Atlantis as shown in Figure 6.3.

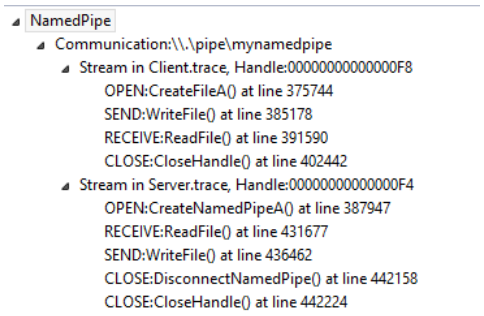
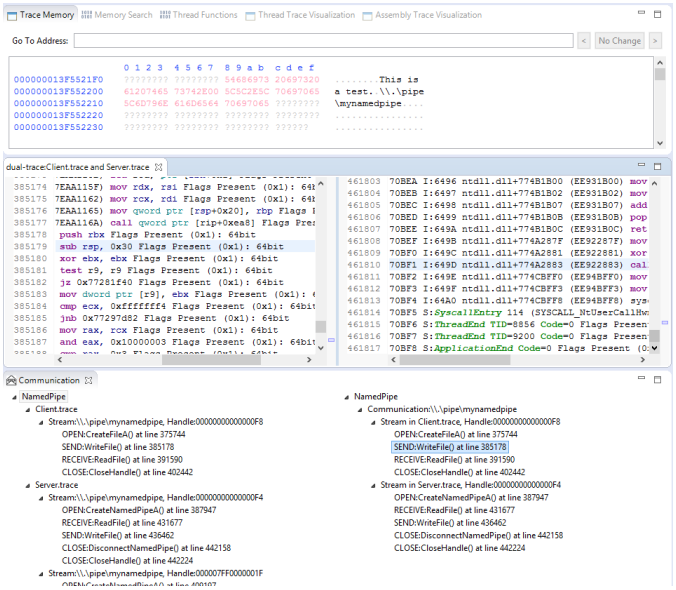
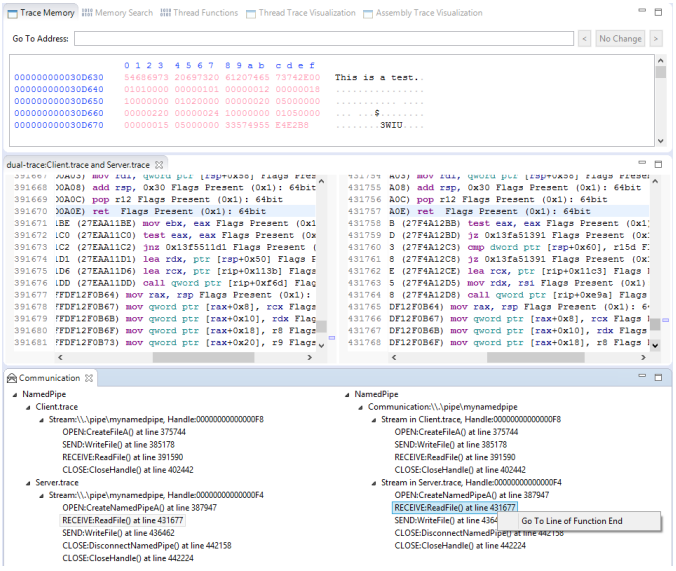



Figure 6.3: Identified Communication of $dual\_trace\_1$

After I got the identified communication from $dual\_trace\_1$. I navigated from the send event entries via "double click" and receive event entries via "Go To Line of Function End" to the traces. The navigation results are shown is Figure 6.4 and 6.5.



(a) Client Send Event Navigation



(b) Server Receive Event Navigation

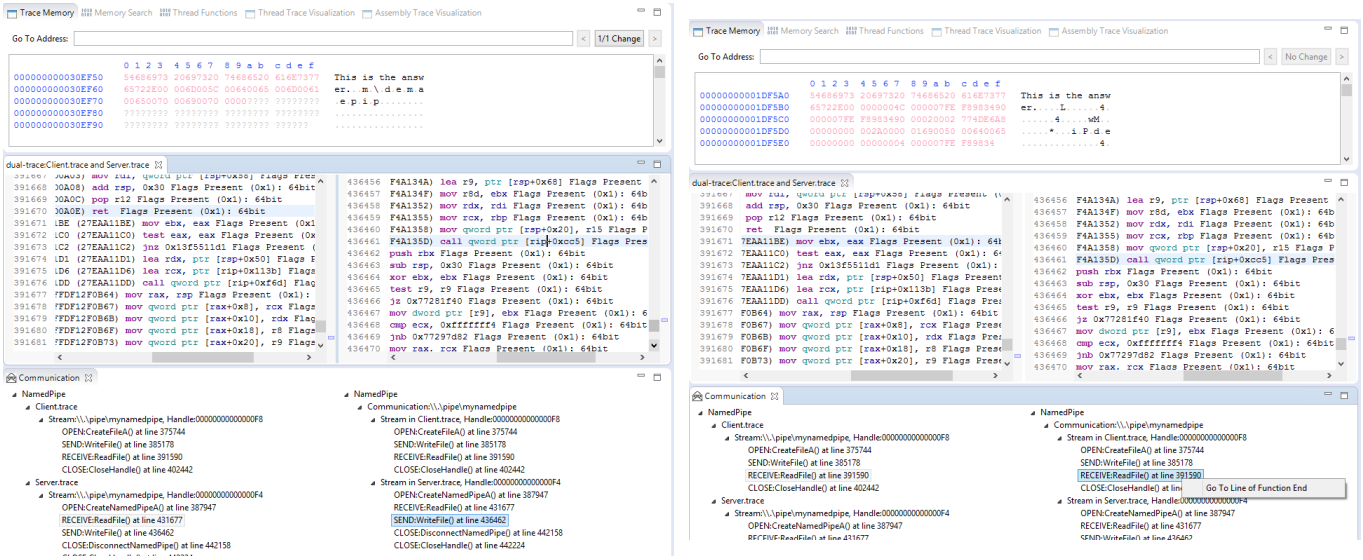
Figure 6.4: Navigation Results for the Transmitted Message "$This\ is\ a\ test.$"

Figure 6.4a shows when I double clicked on the $WriteFile$ function call event of the $Client.trace$, it brought me to the "Trace view" of the $Client.trace$ on line 385178 where the function started, and the "Trace Memory view" jumped to the memory address $0x13F5521F8$, which is the address for the send buffer of the message "$This\ is\ a\ test.$".

Figure 6.4b shows when I selected "Go To Line of Function End" in the right click menu on the $ReadFile$ function call event of $Server.trace$, it brought me to the "Trace view" of $Server.trace$

on line 431757 where the function returned, and the "Trace Memory view" jumped to the memory address $0x30D630$, which is the address for the receive buffer of the message *"This is a test."*.

These two figures show how the message *"This is a test."* being transmitted from the client to the server.



(a) Server Send Event Navigation

(b) Client Receive Event Navigation

Figure 6.5: Navigation Results for the Transmitted Message *"This is the answer."*

Figure 6.5a shows when I double clicked on the $WriteFile$ function call event of $Server.trace$, it brought me to the "Trace view" of $Server.trace$ on line 436462 where the function started, and the "Trace Memory view" jumped to the memory address $0x30EF50$, which is the address for the send buffer of the message *"This is the answer."*.

Figure 6.5b shows when I selected "Go To Line of Function End" in the right click menu on the $ReadFile$ function call event of the $Client.trace$, it brought me to the "Trace view" of the $Client.trace$ on line 391670 where the function returned, and the "Trace Memory view" jumped to the memory address $0x1DF5A0$, which is the address for the receive buffer of the message *"This is the answer."*.

This two figures perfectly show how the message *"This is the answer."* being transmitted from the server to the client.

## 6.2 Experiment 2

### 6.2.1 Experiment Design

In the second experiment, a program was run as a named pipe server. In this server program, four named pipes were created and could be connected by up to four clients at a time. Two other programs as the named pipe clients actually connected to this server. Those two clients (client 1 and client 2) used the identical executable program but started in sequence, one after the other. Figure 6.6 is the sequence diagram of the interaction among the server and the two clients. This sequence diagram only exemplify a possible sequence of the events. The actual events sequence can vary depending on the run-time environment.

Three traces were captured at the time when these three programs were running and interacting. They are $Server.trace$ for the program execution of server , $Client1.trace$ for the program execution of Client 1 and $Client2.trace$ for the program execution of Client 2. These three traces were analyzed as two dual_traces. The one consists of $Server.trace$ and $Client1.trace$ is named $dual\_trace\_21$. The other consists of $Server.trace$ and $Client2.trace$ is named as $dual\_trace\_22$. I performed the "Stream Extraction" and "Communication Identification" operations for these two dual_traces with the function descriptor in Table 6.4.

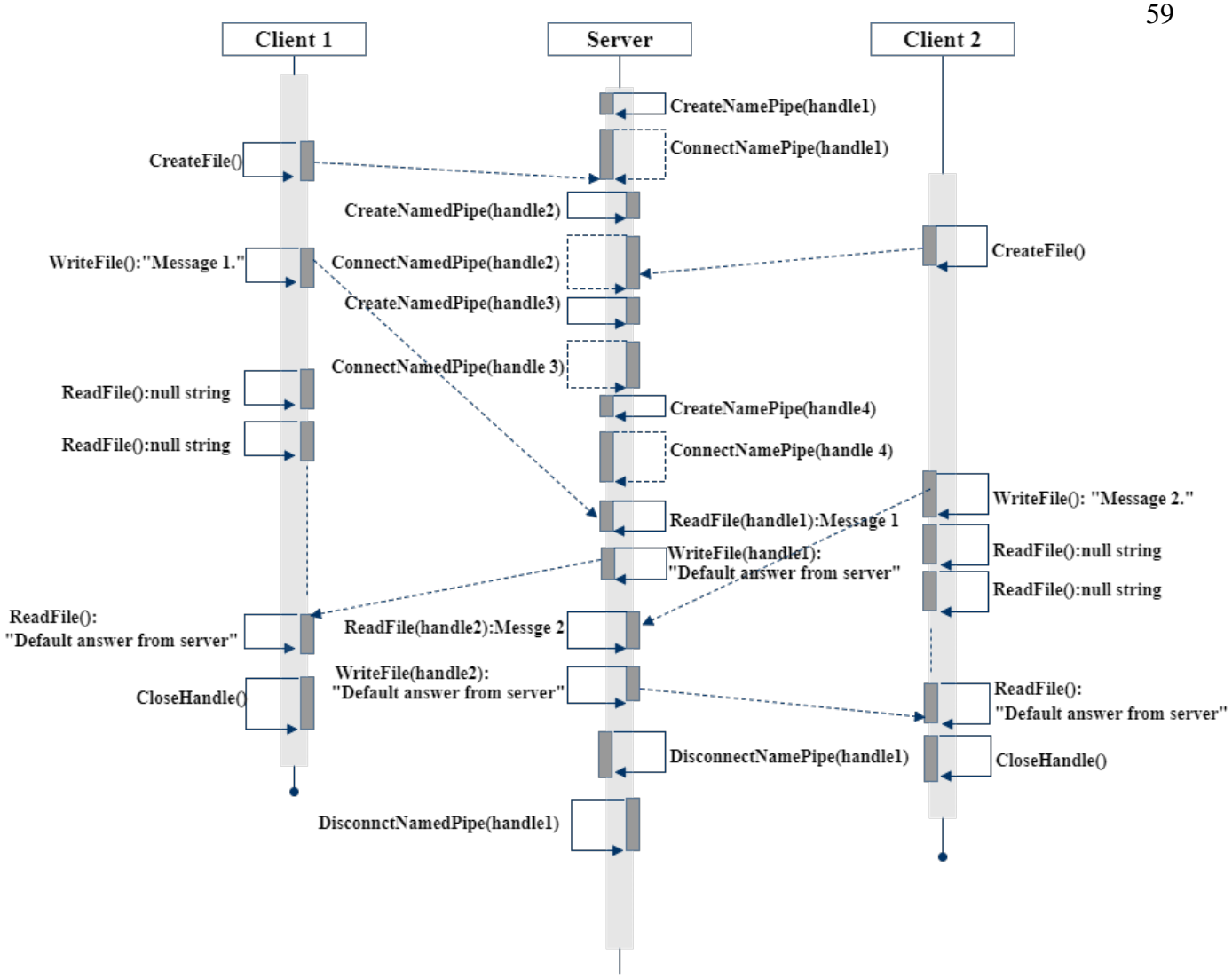



Figure 6.6: Sequence Diagram of Experiment 2

Table 6.4: Function Descriptor of Named Pipe for Experiment 2

| Name | Type | Input Parameters Description | | | Output Parameters Description | | |
|---|---|---|---|---|---|---|---|
| | | Name | Register | Addr/Val | Name | Register | Addr/Val |
| CreateNamedPipe | open | FileName | RCX | Addr | Handle | RAX | Val |
| CreateFile | open | FileName | RCX | Addr | Handle | RAX | Val |
| WriteFile | send | Handle | RCX | Val | Length | R9 | Val |
| | | SendBuf | RDX | Addr | RetVal | RAX | Val |
| ReadFile | receive | Handle | RCX | Val | Length | R9 | Val |
| | | RecvBuf | RDX | Addr | RetVal | RAX | Val |
| GetOverlappedResult | receive | Handle | RCX | Val | OverlapStruct | RDX | Addr |
| | | | | | RetVal | RAX | Val |
| CloseHandle | close | Handle | RCX | Val | RetVal | RAX | Val |
| DisconnectNamedPipe | close | Handle | RCX | Val | RetVal | RAX | Val |

### 6.2.2 Dual_trace Analysis Result Walk Through

In this section, I will first walk through the function call event reconstruction and stream extraction results for each traces: $Server.trace$, $Client1.trace$ and $Client2.trace$. Then I will walk through the communication identification results for each dual_traces: $dual\_trace\_21$ and $dual\_trace\_22$.

***Server.trace* :**

$Server.trace$ has 1789627 instruction lines. Twelve function call events were reconstructed from this trace as listed in Table 6.3.

Table 6.5: The sequence of function call events of $Server.trace$

| Line | Event |
|---|---|
| 1732413 | $funN: CreateNamedPipeA, type: open, inparams: \{Handle: 0x118, FileName: \text{“}\backslash pipe\backslash mynamepipe\text{”}\}, outparams: \{RetVal: 0\}$ |
| 1741477 | $funN: CreateNamedPipeA, type: open, inparams: \{Handle: 0x120, FileName: \text{“}\backslash pipe\backslash mynamepipe\text{”}\}, outparams: \{RetVal: 0\}$ |
| 1749553 | $funN: CreateNamedPipeA, type: open, inparams: \{Handle: 0x128, FileName: \text{“}\backslash pipe\backslash mynamepipe\text{”}\}, outparams: \{RetVal: 0\}$ |
| 1757626 | $funN: CreateNamedPipeA, type: open, inparams: \{Handle: 0x130, FileName: \text{“}\backslash pipe\backslash mynamepipe\text{”}\}, outparams: \{RetVal: 0\}$ |
| 1765903 | $funN: GetOverlappedResult, type: receive, inparams: \{Handle: 0x118\}, outparams: \{OverlapStruct: \text{“”}, RetVal: 0\}$ |
| 1765950 | $funN: ReadFile, type: receive, inparams: \{Handle: 0x118\}, outparams: \{RecvBuf: \text{“}Message\ 2\text{”}, Length: 10, RetVal: 0\}$ |
| 1770738 | $funN: WriteFile, type: send, inparams: \{Handle: 0x118, SendBuf: \text{“}Default\ answer\ from\ server\text{”}\}, outparams: \{Length: 20\}$ |
| 1771629 | $funN: GetOverlappedResult, type: receive, inparams: \{Handle: 0x120\}, outparams: \{OverlapStruct: \text{“”}, RetVal: 0\}$ |
| 1771676 | $funN: ReadFile, type: receive, inparams: \{Handle: 0x120\}, outparams: \{RecvBuf: \text{“}Message\ 1\text{”}, Length: 10, RetVal: 0\}$ |
| 1775507 | $funN: WriteFile, type: send, inparams: \{Handle: 0x120, SendBuf: \text{“}Default\ answer\ from\ server\text{”}\}, outparams: \{Length: 20\}$ |
| 1777180 | $funN: DisconnectNamedPipe, type: close, inparams: \{Handle: 0x118\}, outparams: \{RetVal: 0\}$ |
| 1778658 | $funN: DisconnectNamedPipe, type: close, inparams: \{Handle: 0x120\}, outparams: \{RetVal: 0\}$ |

There are four handle values in this event sequence: $0x118$, $0x120$, $0x128$, $0x130$. So four

streams are extracted with these four handle identifiers. Both stream $0x118$ and $0x120$ have five events while stream $0x128$ and $0x130$ only have one channel open event. The extracted streams were listed in the left table of "Communication view" as shown in Figure 6.7.

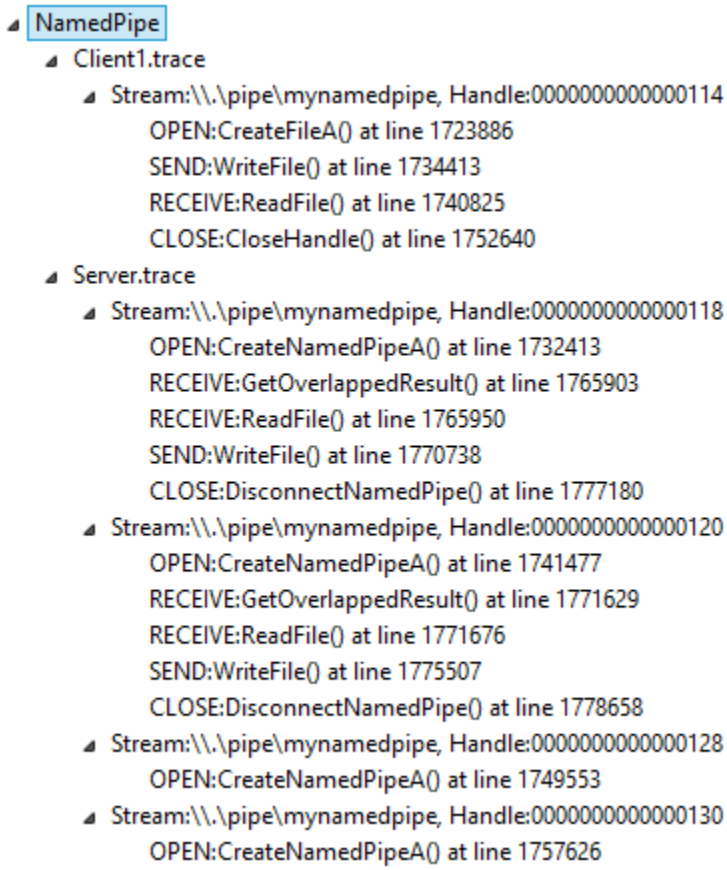



Figure 6.7: Extracted Streams of $dual\_trace\_21$

### $Client1.trace$ :

$Client1.trace$ has 1764281 instruction lines. Four function call events were reconstructed from this trace as listed in Table 6.6.

Table 6.6: The sequence of function call events of $Client1.trace$

| Line | Event |
|---|---|
| 1723886 | $funN : CreateFileA, type : open, inparams : \{Handle : 0x114, FileName : \text{“}\backslash pipe\backslash mynamepipe\text{”}\}, outparams : \{RetVal : 0\}$ |
| 1734413 | $funN : WriteFile, type : send, inparams : \{Handle : 0x114, SendBuf : \text{“}Message\ 1\text{”}\}, outparams : \{Length : 15\}$ |
| 1740825 | $funN : ReadFile, type : receive, inparams : \{Handle : 0x114\}, outparams : \{RecvBuf : \text{“}Default\ answer\ from\ server\text{”}, Length : 17, RetVal : 0\}$ |
| 1752640 | $funN : CloseHandle, type : close, inparams : \{Handle : 0x114\}, outparams : \{RetVal : 0\}$ |

All the values of the handle parameter of these four events are $0x114$. So that a stream identified

by the handle $0x114$ was extracted, which consists of all these four function call events. The extracted stream was listed in the left table of "Communication view" as shown in Figure 6.7.

### $Client2.trace$ :

$Client2.trace$ has 1764441 instruction lines. Four function call events were reconstructed from this trace as listed in Table 6.7.

Table 6.7: The sequence of function call events of $Client2.trace$

| Line | Event |
|---|---|
| 1723886 | $funN : CreateFileA, type : open, inparams : \{Handle : 0x114, FileName : \text{“}\backslash pipe\backslash mynamepipe\text{”}\}, outparams : \{RetVal : 0\}$ |
| 1734413 | $funN : WriteFile, type : send, inparams : \{Handle : 0x114, SendBuf : \text{“}Message\ 1\text{”}\}, outparams : \{Length : 15\}$ |
| 1740825 | $funN : ReadFile, type : receive, inparams : \{Handle : 0x114\}, outparams : \{RecvBuf : \text{“}Default\ answer\ from\ server\text{”}, Length : 17, RetVal : 0\}$ |
| 1752800 | $funN : CloseHandle, type : close, inparams : \{Handle : 0x114\}, outparams : \{RetVal : 0\}$ |

All the values of the handle parameter of these four events are $0x114$. So that a stream identified by the handle $0x114$ was extracted, which consists of all these four function call events.

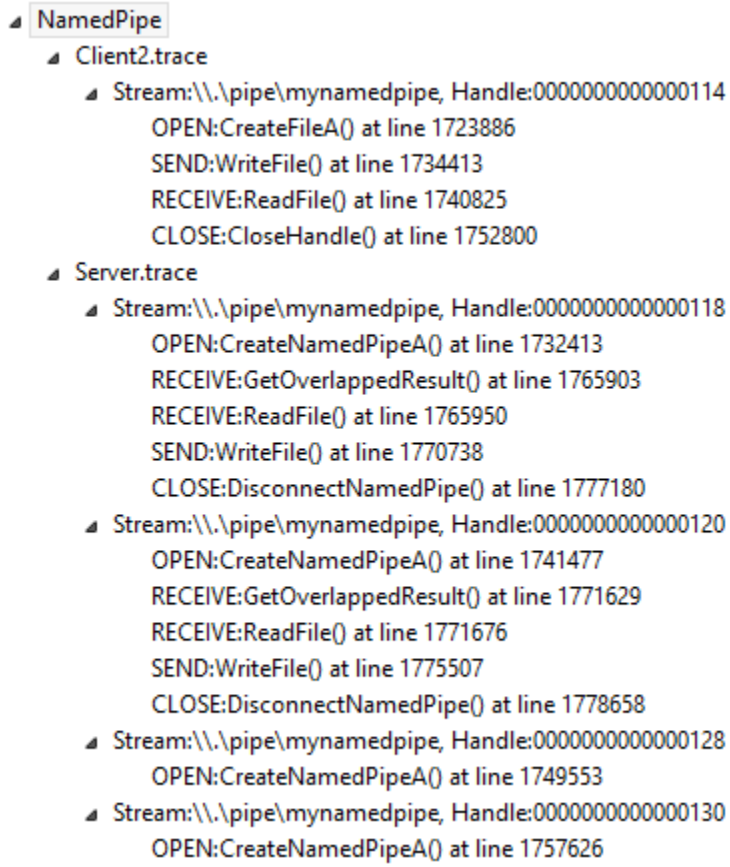



Figure 6.8: Extracted Streams of $dual\_trace\_22$

***Dual_trace_21 :***

The value of the $FileName$ parameter of the $CreateFileA$ function call event in $Client1.trace$ is "$.\backslash pipe\backslash mynamepipe$" as shown in Table 6.6. Meanwhile, the values of the $FileName$ parameter of the $CreateNamePipeA$ function call events of all streams in $Server.trace$ are also "$.\backslash pipe\backslash mynamepipe$" as shown in Table 6.5. So that, the stream identified by the handle $0x114$ of $Client.trace$ is matched to all the streams of $Server.trace$.

The send and receive packet contents and the payload concatenation of the streams in the server and client 1 are listed in Table 6.8. Comparing the concatenation of each streams in $Server.trace$, it is obvious that the send payload concatenation of Stream $0x114$ in $Client1.trace$ equals to the receive payload concatenation of Stream $0x120$ in $Server.trace$, while on the other direction, the send payload concatenation of Stream $0x120$ in $Server.trace$ equals to the receive payload concatenation of Stream $0x114$ in $Client1.trace$. So that, only Stream $0x120$ of $Server.trace$ and Stream $0x114$ of $Client1.trace$ satisfy the content preservation of the reliable communication.

Table 6.8: Content Summarize of the Extracted Streams

| | Handle | Receive | | Send | |
|---|---|---|---|---|---|
| | | Events | Concatenation | Events | Concatenation |
| ***Server.trace*** | $0x118$ | $GetOverlappedResult$ : "" | "$Message\ 2$" | $WriteFile$ : "$Default\ answer\ from\ server$" | "$Default\ answer\ from\ server$" |
| | | $ReadFile$ : "$Message\ 2$" | | | |
| | $0x120$ | $GetOverlappedResult$ : "" | "$Message\ 1$" | $WriteFile$ : "$Default\ answer\ from\ server$" | "$Default\ answer\ from\ server$" |
| | | $ReadFile$ : "$Message\ 1$" | | | |
| | $0x128$ | - | - | - | - |
| | $0x130$ | - | - | - | - |
| ***Client1.trace*** | $0x114$ | $ReadFile$ : "$Default\ answer\ from\ server$" | "$Default\ answer\ from\ server$" | $WriteFile$ : "$Message\ 1$" | "$Message\ 1$" |

Therefore, Stream $0x120$ of $Server.trace$ and Stream $0x114$ of $Client1.trace$ are eventually output as a communication by the "Communication Identification" operation in the right table of "Communication view" as shown in Figure 6.9.

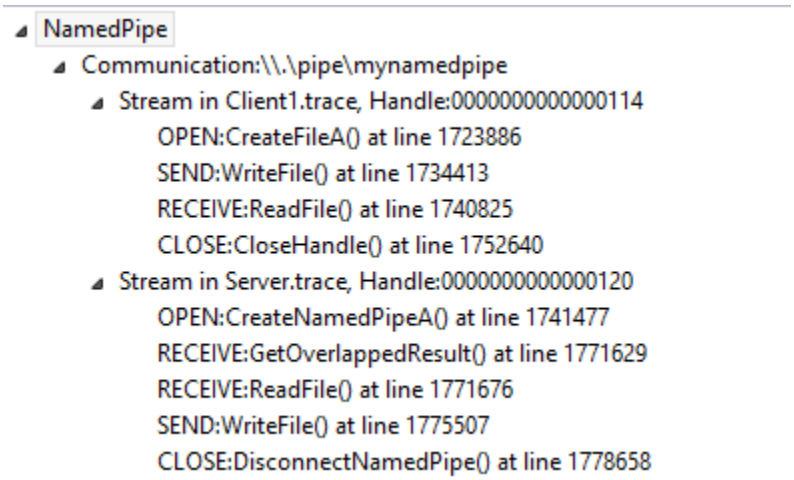



Figure 6.9: Identified Communication of $dual\_trace\_21$

After I got the identified communication from $dual\_trace\_21$. I navigated from the send and receive events back to the traces. The navigation results are shown is the Figure 6.10, Figure 6.11 and Figure 6.12.

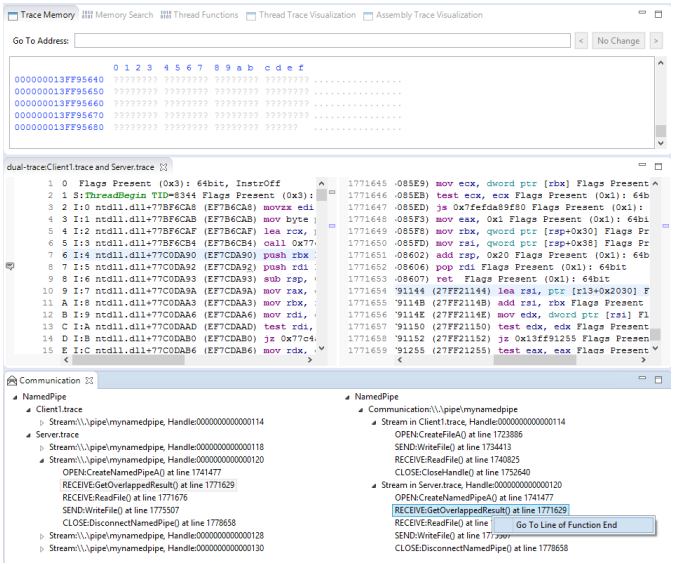
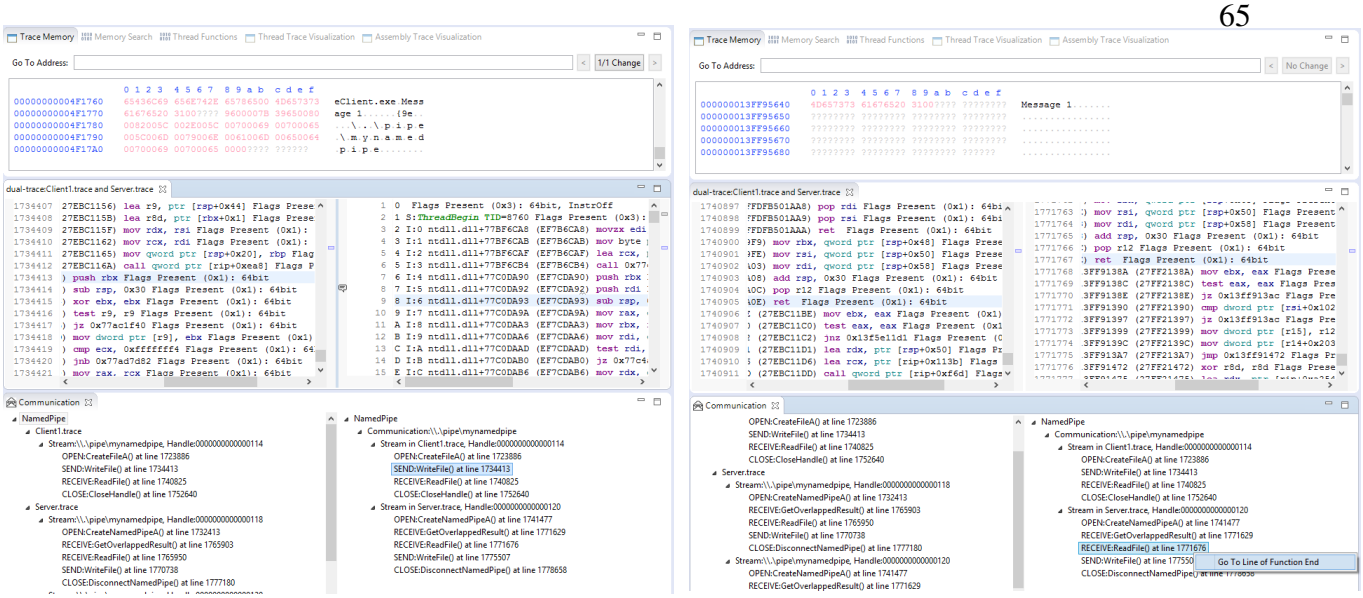



Figure 6.10: Navigation Result for the Function Call Event: $GetOverlappedResult$

Figure 6.10 shows when I selected "Go To Line of Function End" in the right click menu on the $GetOverlappedResult$ function call event of the $Server.trace$, it brought me to the "Trace view" of the $Server.trace$ on line 1771653 where the function returned. However, since this function call didn't get any message, the "Trace Memory view" is blank.



(a) Client 1 Send Event Navigation

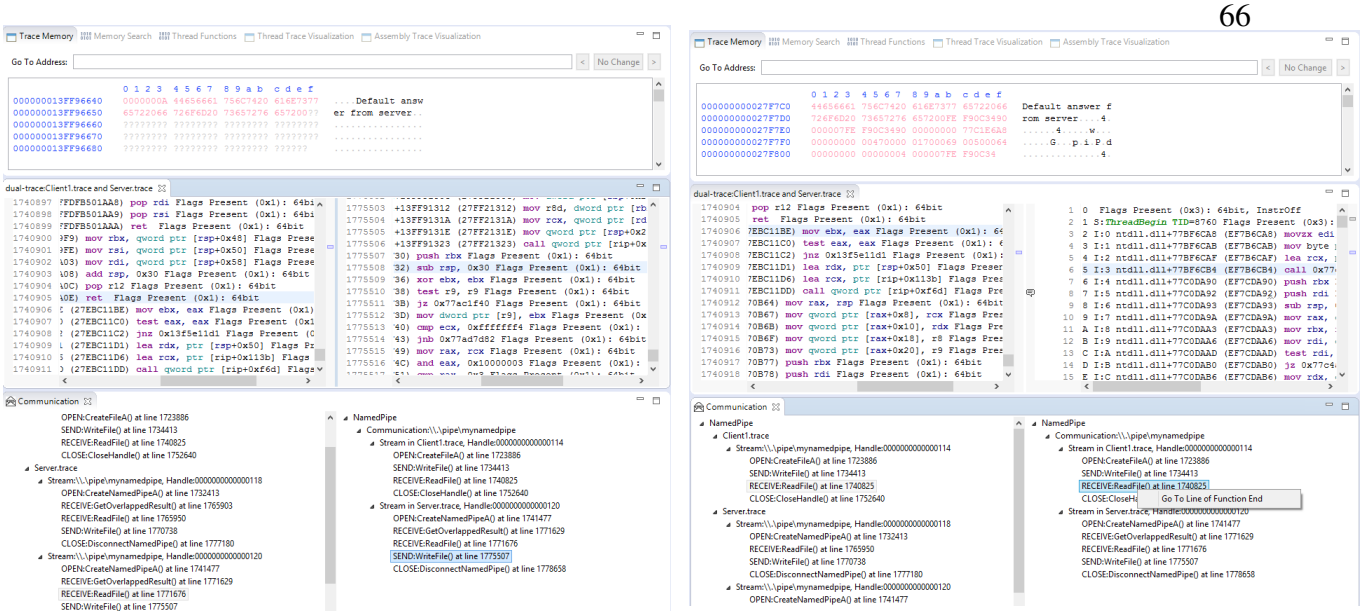
(b) Sever Receive Event Navigation

Figure 6.11: Navigation Results for the Transmitted Message "$Message\ 1$"

Figure 6.11a shows when I double clicked on the $WriteFile$ function call event of $Client.trace$, it brought me to the "Trace view" of $Client.trace$ on line 1734413 where the function started, and the "Trace Memory view" jumped to the memory address $0x4F176c$, which is the address for the send buffer of the message "$Message\ 1$".

Figure 6.11b shows when I selected "Go To Line of Function End" in the right click menu on the $ReadFile$ function call event of the $Server.trace$, it brought me to the "Trace view" of the $Server.trace$ on line 1771767 where the function returned, and the "Trace Memory view" jumped to the memory address $0x13FF95640$, which is the address for the receive buffer of the message "$Message\ 1$".

This two figures perfectly show how the message "$Message\ 1$" being transmitted from the client 1 to the server.



(a) Server Send Event Navigation

(b) Client 1 Receive Event Navigation

Figure 6.12: Navigation Results for the Transmitted Message "$Default\ answer\ from\ server$"

Figure 6.12a shows when I double clicked on the $WriteFile$ function call event of $Server.trace$, it brought me to the "Trace view" of $Server.trace$ on line 1775507 where the function started, and the "Trace Memory view" jumped to the memory address $0x30EF50$, which is the address for the send buffer of the message "$Default\ answer\ from\ server$".

Figure 6.12b shows when I selected "Go To Line of Function End" in the right click menu on the $ReadFile$ function call event of $Client.trace$, it brought me to the "Trace view" of $Client.trace$ on line 1740905 where the function returned, and the "Trace Memory view" jumped to the memory address $0x1DF5A0$ of the receive buffer of the message "$Default\ answer\ from\ server$".

This two figures perfectly show how the message "$Default\ answer\ from\ server$" being transmitted from the server to the client 1.

### $Dual\_trace\_22$ :

Same as $dual\_trace\_22$, all streams of $Server.trace$ will be matched to the only one stream of $Client2.trace$ by the stream matching algorithm.

The send and receive packet contents and the payload concatenation of the streams in the server and client 1 are listed in Table 6.9. Comparing the concatenation of each streams in $Server.trace$, it is obvious that the send payload concatenation of Stream $0x114$ in $Client2.trace$ equals to the receive payload concatenation of Stream $0x118$ in $Server.trace$, while on the other direction,

the send payload concatenation of Stream $0x118$ in $Server.trace$ equals to the receive payload concatenation of Stream $0x114$ in $Client2.trace$. So that, only Stream $0x118$ of $Server.trace$ and Stream $0x114$ of $Client2.trace$ satisfy the content preservation of the reliable communication.

Table 6.9: Content Summarize of the Extracted Streams

| | Handle | Receive | | Send | |
|---|---|---|---|---|---|
| | | Events | Concatenation | Events | Concatenation |
| *Server.trace* | $0x118$ | $GetOverlappedResult$ : "" | "$Message\ 2$" | $WriteFile$ : "$Default\ answer\ from\ server$" | "$Default\ answer\ from\ server$" |
| | | $ReadFile$ : "$Message\ 2$" | | | |
| | $0x120$ | $GetOverlappedResult$ : "" | "$Message\ 1$" | $WriteFile$ : "$Default\ answer\ from\ server$" | "$Default\ answer\ from\ server$" |
| | | $ReadFile$ : "$Message\ 1$" | | | |
| | $0x128$ | - | - | - | - |
| | $0x130$ | - | - | - | - |
| *Client2.trace* | $0x114$ | $ReadFile$ : "$Default\ answer\ from\ server$" | "$Default\ answer\ from\ server$" | $WriteFile$ : "$Message\ 2$" | "$Message\ 2$" |

Therefore, Stream $0x118$ of $Server.trace$ and Stream $0x114$ of $Client2.trace$ are eventually output as a communication by the "Communication Identification" operation in the right table of "Communication view" as shown in Figure 6.13.

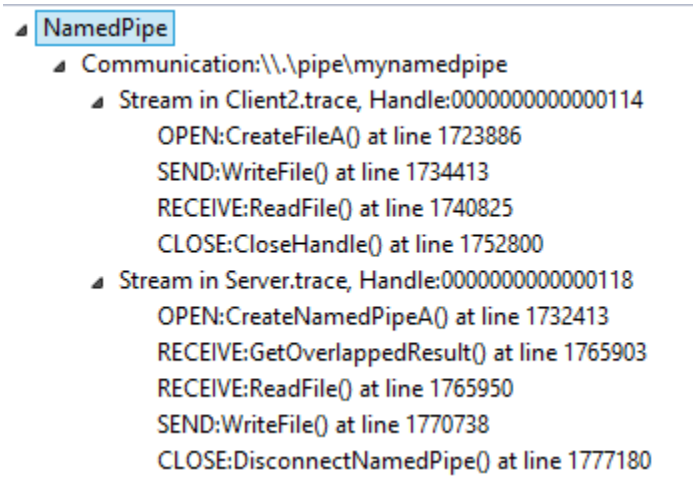



Figure 6.13: Identified Communication of $dual\_trace\_22$

After I got the identified communication from $dual\_trace\_22$ I navigated from the send and receive events back to the traces. The navigation results are shown is the Figure 6.14, Figure 6.15 and Figure 6.16.

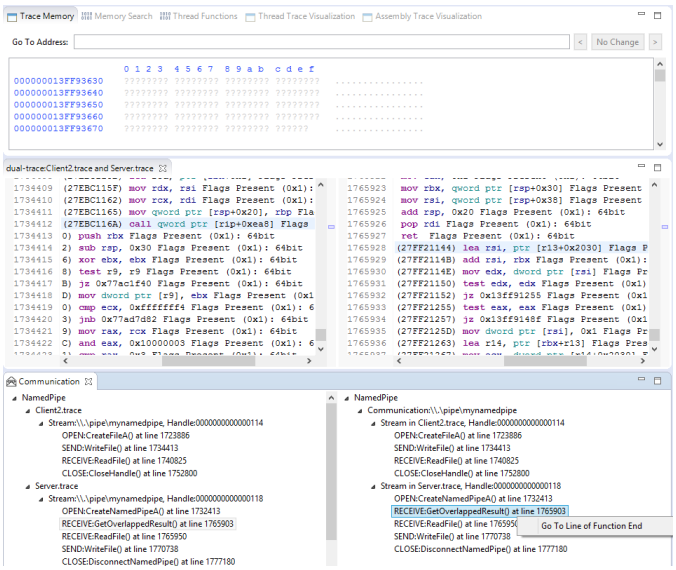
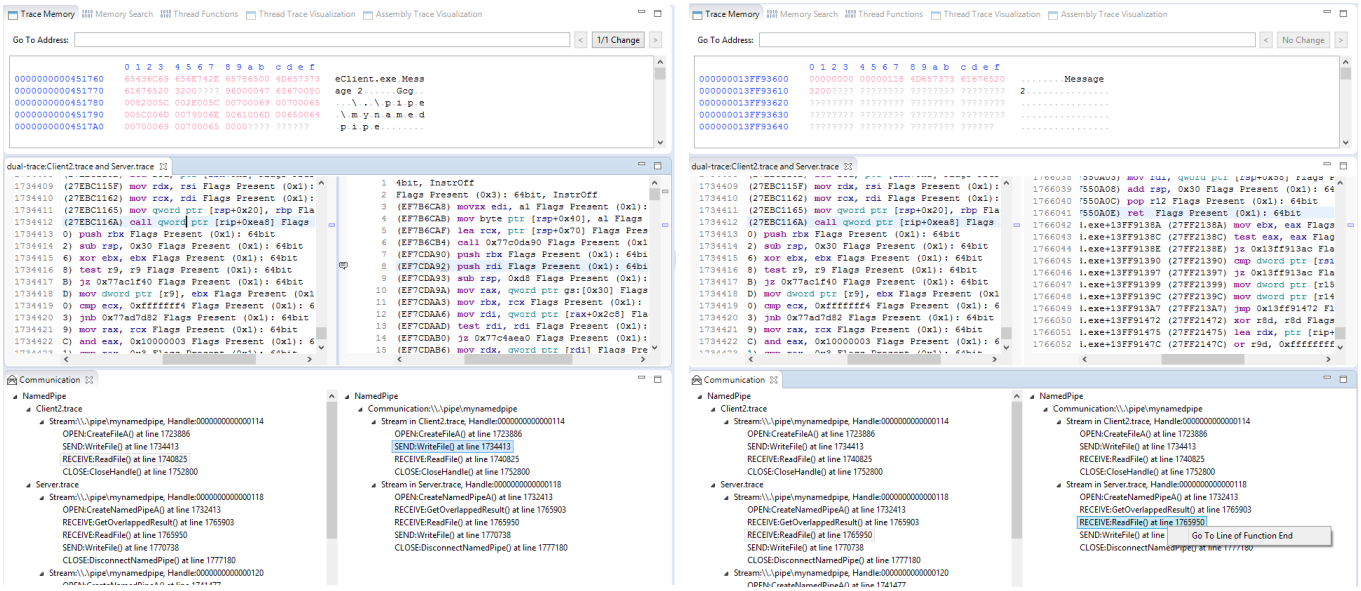



Figure 6.14: Navigation Result for the Function Call Event: *GetOverlappedResult*

Figure 6.14 shows when I selected "Go To Line of Function End" in the right click menu on the *GetOverlappedResult* function call event of the *Server.trace*, it brought me to the "Trace view" of the *Server.trace* on line 1765927 where the function returned. However, since this function call didn't get any message, the "Trace Memory view" is blank.




(a) Client 2 Send Event Navigation

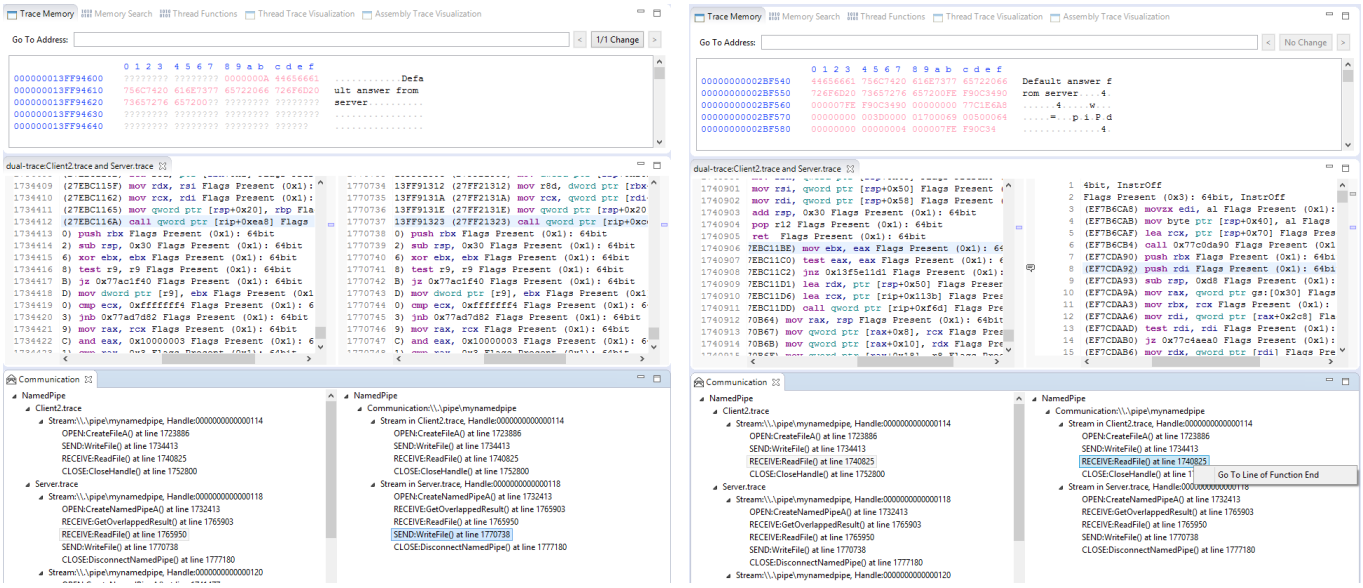
(b) Sever Receive Event Navigation

Figure 6.15: Navigation Results for the Transmitted Message "*Message* 2"

Figure 6.15a shows when I double clicked on the $WriteFile$ function call event of $Client.trace$, it brought me to the "Trace view" of $Client.trace$ on line 1734413 where the function started, and the "Trace Memory view" jumped to the memory address $0x45176c$, which is the address for the send buffer of the message "$Message\ 2$".

Figure 6.15b shows when I selected "Go To Line of Function End" in the right click menu on the $ReadFile$ function call event of the $Server.trace$, it brought me to the "Trace view" of the $Server.trace$ on line 1766041 where the function returned, and the "Trace Memory view" jumped to the memory address $0x13FF93608$, which is the address for the receive buffer of the message "$Message\ 2$".

These two figures show how the message "$Message\ 2$" being transmitted from the client 2 to the server.



(a) Server Send Event Navigation

(b) Client 2 Receive Event Navigation

Figure 6.16: Navigation Results for the Transmitted Message "$Default\ answer\ from\ server$"

Figure 6.16a shows when I double clicked on the $WriteFile$ function call event of the $Server.trace$, it brought me to the "Trace view" of the $Server.trace$ on line 1770738 where the function started, and the "Trace Memory view" jumped to the memory address $0x13FF94608$, which is the address for the send buffer of the message "$Default\ answer\ from\ server$".

Figure 6.16b shows when I selected "Go To Line of Function End" in the right click menu on the $ReadFile$ function call event of $Client.trace$, it brought me to the "Trace view" of $Client.trace$ on line 1740906 where the function returned, and the "Trace Memory view" jumped to the memory address $0x2BF540$ of the receive buffer of the message "$Default\ answer\ from\ server$".

This two figures perfectly show how the message "$Default\ answer\ from\ server$" being transmitted from the server to the client 2.

## 6.3 Conclusion

By walking through the analysis results of the "Stream Extraction" and "Communication Identification" operations in these two experiments, I can conclude that "Stream Extraction" operation is capable of properly extracting the streams from both traces in a dual_trace, while "Communication Identification" operation is capable of identifying the communication between the two traces of a dual_trace.

Moreover, from the "Communication view", the user can easily navigate back to the exact instruction line where the function start or end. The messages being transmitted can be shown in the "Trace Memory view" accurately.

The experiments are restricted by the traces that can be captured. The current in-house tracer of DRDC is only able to capture the function information for some .dll files. kernel32.dll which contains the functions for Named pipe is one of the .dll files that this tracer can capture. Functions for the other communication methods discussed in this thesis can not be captured by DRDC's tracer currently. Thurs both of the conducted experiments used the named pipe communication method.

These two experiments are not provided as an empirical evaluation but it shows the usefulness of the algorithms and the prototype implementation.

# Chapter 7

# Conclusions and Future Work

This thesis illustrates a 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 provides more reliable information for vulnerability detection and program analysis.

In this thesis, I presented an approach to analyze two traces to understand how they communicate with each other. I first defined a communication model. This abstract model depicts the outline of a communication between two running programs, which gives the ground rules for the communication analysis. Then I presented the formalization of the dual_trace. The formalization indicates that all traces comply to it can be used to conducted the communication analysis.

I also developed the algorithms for the communication identification. The developed algorithms not only solve the problem for specific communication methods but also provide clear and referable examples to develop other algorithms for other 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. This feature provides the users a way to extend their concerned communication methods through the configuration file. The 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. The experiments conducted in this work demonstrate the feasibility and usability of the model and the algorithms.

Future work of this research includes:

- Extend the model to be more generalize for all kinds of interaction but not only the message transferring communications, for example remote procedure call

- Visualize the communications identified from the dual_trace (A sequence diagram might be a good choice to illustrate all the events in the traces and the matched events from both traces.)
- Conduct user studies of the communication analysis approach and the prototype
- Conduct an empirical study to properly evaluate the algorithms and implementation presented in this thesis
- Apply the communication analysis approach developed in this thesis on the practical problems. for example, the analysis of Inter-Component communication of Android.

# Appendix A

# Microsoft x64 Calling Convention for C/C++

- RCX, RDX, R8, R9 are used for integer and pointer arguments in that order left to right.
- XMM0, 1, 2, and 3 are used for floating point arguments.
- Additional arguments are pushed on the stack left to right. ...
- Parameters less than 64 bits long are not zero extended; the high bits contain garbage.
- Integer return values (similar to x86) are returned in RAX if 64 bits or less.
- Floating point return values are returned in XMM0.
- 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 B

# Function Descriptor Configuration file Example

Listing B.1: communicationMethods.json

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

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

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

# Appendix C

# 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.

## C.1 The Editor Area Split Handler

Listing C.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;
      }
}
```

## C.2 Get the Active Parallel Editors

Listing C.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 D

# Code of the Programs in the Experiments

## D.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 D.1: NamedPipeServer.cpp

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

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

#define BUFSIZE 512

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
            }
      }

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

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

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

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

Listing D.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;
}
```

## D.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 D.3: NamedPipeServerOverlapped.cpp

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

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

unsigned int ReplyCount = 0;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
                        continue;
                    }

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

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

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

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

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

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

                default:
                {
                    return 0;
                }
                }
        }

        return 0;
}

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

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

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

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

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

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

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

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

Listing D.4: test.bat

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

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