Identifying Communications of Running Programs through Their 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 Thesis 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 communications between programs can help software security engineers understand the behaviour of a system and detect vulnerabilities in a system. Assembly-level execution traces are used for this purpose for two reasons: 1) lack of source code of the running programs, and 2) assembly-level execution traces provide the most accurate run-time behaviour information. In this thesis, I present a communication analysis approach using such execution traces. I first model the message based communication in the context of trace analysis. Then I develop a method and the necessary algorithms to identify 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 show 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. Vendors usually comply with their own quality assurance measures which focus on marketable concerns while leaving security to a lower priority or even worse, they totally ignore it. Therefore, fully relying on the vendor of the software to secure your system and data is impractical and risky. [6]

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 the 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 users of the software from putting their sensitive data and business resources at risk. [6]

Most software vulnerabilities are stimulated by malicious data, and it is valuable to understand how this malicious data triggers the unexpected behaviours. 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 a service or functionality. In these situations, the malicious data might have passed through multiple components and be modified before it reaches the vulnerable point and ultimately triggers an exploitable condition. 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 a security review. [6]

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

this data flow can affect the analysis results greatly.

In this research, I develop a method to identify communications between programs by analysing assembly-level execution traces. This method can guide security engineers in their investigation of the programs' communications through assembly execution traces. 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 our research partner DRDC (Defence Research and Development Canada), for visualizing multiple assembly-level traces to assist their software security analysis. The literature review and conversations with DRDC help to clarify the goal and guided this research. In this section, I discuss the need for performing assembly-level trace investigation for communication analysis. First I explain why security engineers perform assembly-level trace analysis. Then I elaborate why they need to perform communication analysis at the assembly-level trace level.

1.1.1 Why Assembly-level Trace Analysis

Dynamic analysis of programs is adopted mainly in software maintenance and security auditing [23, 3, 18]. Sanjay Bhansali et al. claim that program execution traces with the most intimate detail of a program's dynamic behaviour can facilitate program optimization and failure diagnosis [1]. Jonas Trümper et al. give an example of how tracing can facilitate software-maintenance tasks [20].

Dynamic analysis can be done using debuggers, however, a debugger halts the execution of the system and results in a distortion of the timing behaviour [20]. Instead, tracing a running program with instrumentation provides more accurate run-time behaviour information about the system.

The instrumentation 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 [6]. In this case, assembly-level tracing is the only option to review the security the software.

On the other hand, the binary code is what runs on the system, so binary tracing is more representative for software security engineers than the source code in the terms of auditing. Some

bugs might appear because of a compilation problem or because the compiler optimized 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 resetting the password before the program end would be optimized by the compliers if they implement 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 [9]. This will make the user's password stay 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-level Traces

Programs nowadays do not always work in isolation. 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 sniffers, such as Wireshark[4] and Tcpdump[19], 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~~innocent. Therefore, analysing the communications with transmitted data in instruction and memory access level is a solid way to evaluate the security of a system.

Wen et al. argue that "fuzz testing and symbolic execution are widely applied to detect vulnerabilities in network protocol implementations. Their work focuses on designing a model that guides the symbolic execution for fuzz testing" [21] but ignores the analysis of the output, the execution traces. Furthermore, their work focuses only on the network protocol implementation but not on general communications.

Besides vulnerabilities detection and security analysis, communication analysis with assembly-level 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 another 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 sent wirelessly 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 time that they trace the execution of the programs.

Overall, communication analysis with assembly-level 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 iterative research process with three abstracted stages. The process is iterative because the implementation changed several times due to changes with the model, and the model was modified based on understanding of details of execution traces gained throughout the implementation.

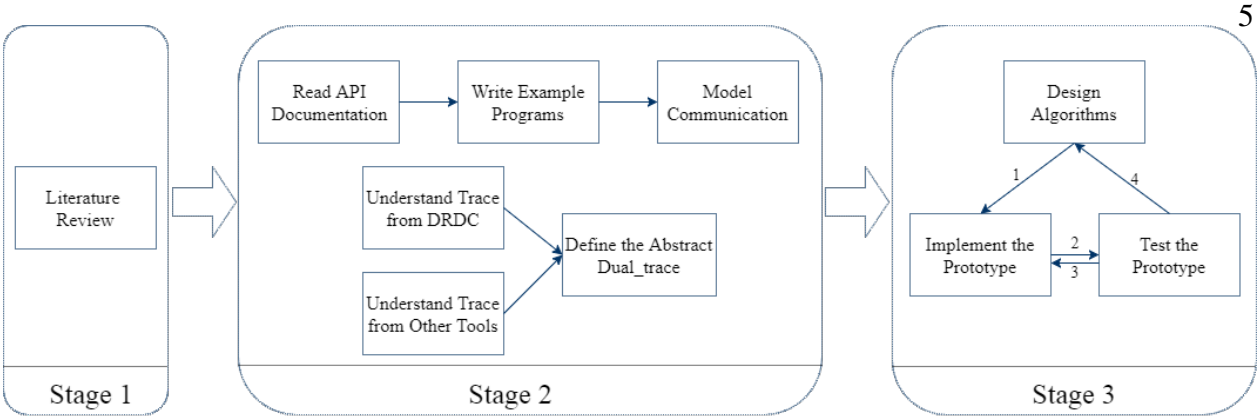

Figure 1.1: Research approach overview

This research requires background knowledge of software security and vulnerabilities. I acquired the background knowledge basically from literature review. It helped me acquire 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-level trace 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 simple example 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-level trace definition was built on the generalization of the trace format provided by our research partner, DRDC. I don't have the access to their home-made assembly-level tracer which is based on PIN[12]. 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-level execution trace. Furthermore, 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 method 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:** A communication model is abstracted from the understanding of several communication methods and is generic to other communication methods. This model indicates how the communication happen in terms of what information of it has been recorded. It can guide the software analyst to analyze the communication. The analyst might be able to retrieve some information of a communication from the traces, such as a sent function call in a trace with matched received function call from the other, or the transmitted messages. However, they might not aware that they can reconstruct the communication with all the essential information as a whole picture.

- **Dual_trace and Functions Descriptor Formalization:**

  By understanding the assembly-level execution traces, a dual_trace was formalized to describe the information that was needed for communication analysis. The dual_trace formalization does not specify the format of the execution traces but defines what information is necessary to fulfil the analysis requirement. All execution traces that comply with this formalization can be used for the analysis. This formalization can be a reference for the design of assembly-level tracer, guiding ~~them~~ what information the tracer should capture to fulfil communication analysis.

  The functions descriptor describes a communication method. Following the functions descriptor formalization, the user can develop a functions descriptor through understanding the mechanism of the communication method.

- **Communication Analysis Approach:** The overall approach to identify the communications in the dual_trace is designed. Eight algorithms were developed for the components in this approach regarding some communication methods or communication types.

- **Prototype:** A prototype for communication analysis through assembly-level execution traces is designed and implemented on Atlantis, which is an assembly-level execution traces analysis environment [11]. Atlantis has many features that benefit the communication analysis such as view synchronization and function inspection. Moreover, the unique memory state reconstruction feature makes the data verification of a communication much easier. After the user understand the communication model and the formalization presented in this thesis, the user can use the old Atlantis (without the implementation of this prototype) to perform the

analysis. However, manually identifying the communication from two traces which might contain millions lines of instructions could be an extremely time consuming and exhausting task. This prototype makes the analysis much more efficient and practical. This prototype also demonstrates that the communication analysis approach is feasible. It is a unique tool for the security engineers to analyze the communications of programs via assembly-level 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 results show the communications are correctly identified.

Finally, in Chapter 7, I conclude the result of this research and outline possible future works.

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 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 [6]. 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 that 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 [8]. These communications can happen

over a 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 while leaving other communication methods such as remote procedural call for feature works.

2.3 Program Execution Tracing at the Assembly-Level

The communication analysis discussed throughout this thesis is based on assembly-level 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 at the assembly instruction level. IDA Pro [7] is a widely used tool in reverse engineering which can capture and analyze system level execution traces. Through open plugin APIs, IDA Pro allows plugin such as Codemap [5] to provide more sufficient features for "run-trace" visualization. PIN [12] is a tool for the 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 [2] and OllyDbg [22] also provide debugging and tracing functionality at the assembly-level.

2.4 Atlantis

Atlantis is a trace analysis environment developed in the Chisel lab at the University of Victoria [11]. It can support analysis for multi-gigabyte assembly-level traces. There are several features that distinguish 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; 2) reconstruction and navigation of system functions and processes; and 3) a powerful search facility to query and navigate traces [11]. The work of this thesis is not an extension of Atlantis. But it takes advantage of Atlantis by reusing its existing features to assist the dual_trace analysis. The reason that I choose Atlantis for communication analysis is not because it was develop by the research group I work in, but the features that it already has make the implementation of the communication analysis prototype much easier than developing a new tool or use some other existing tool such as IDA Pro.

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 the two communication categories: reliable and unreliable communications.

3.1 Communication Methods Categorization

In terms of their reliability of data transmission, the communications can be divided into two categories: reliable and unreliable. ~~A reliable communicationguarantees~~ In a reliable communication, the data being sent by one endpoint through the channel is always received losslessly and in the same order in the other endpoint. ~~However, in~~ In some reliable communication, the sent packets can be re-segmented and arrives at the receiver end. In contrast, an unreliable communication does not guarantee the data being sent always arrives at the receiver end. Moreover, the data packets can arrive in any order. However, the positive side of the unreliable communications is that the packets always arrives as the original packets, no data re-segmentation happens. An endpoint is an instance in a program at which a stream of data is sent, received or both (e.g., a socket handle for TCP or a file handle for the named pipe). A channel is a conduit connecting two endpoints through which data can be sent and received. The categorization here doesn't consider if the physical medium used for the communication is lossless. It stands in the application point of view and see how much reliability of the communication methods the protocol can provide. For example, packets can be loss during the transmission in TCP channels. However, the protocol is designed to try its best to guarantee the losslessness by re-transmission, congestion control, etc. So from the point of

view of the application, all data transmitted is controlled in an orderly fashion, is received in the correct order and is intact. Table 3.1 gives examples of how 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 to start new communications. However, the reopened channel is considered as a new communication. The model is not about how the communication works but what it looks like. There are many communication methods in the real world and they are compatible to this communication definition.

3.2.1 Communication Definition

In the context of a dual_trace, a communication is a sequence of data transmitted between two endpoints through a communication channel. 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, 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.

3.2.2.1 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 given data stream, we define the data in this stream as the concatenation of the payload of all the packets in the order of sending or receiving in this stream, and denote it as $data(d)$.
  Given $d =< pk_1, pk_2, ..., pk_n >$, $data(d) = pl(pk_1) \cdot pl(pk_2) \cdot \ldots \cdot pl(pk_n)$.

For a communication, the received data of an endpoint should always be a prefix of (potentially equal to) the sent data of the other. In other words, for a communication $c =< ch, < h_0, dr_0, ds_0 >, < h_1, dr_1, ds_1 >>$, $data(dr_0)$ 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.

3.2.2.2 Properties of unreliable communication

In an unreliable communication, the properties are not a concern 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, 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.

Figure 3.1 is an example of an reliable communication.

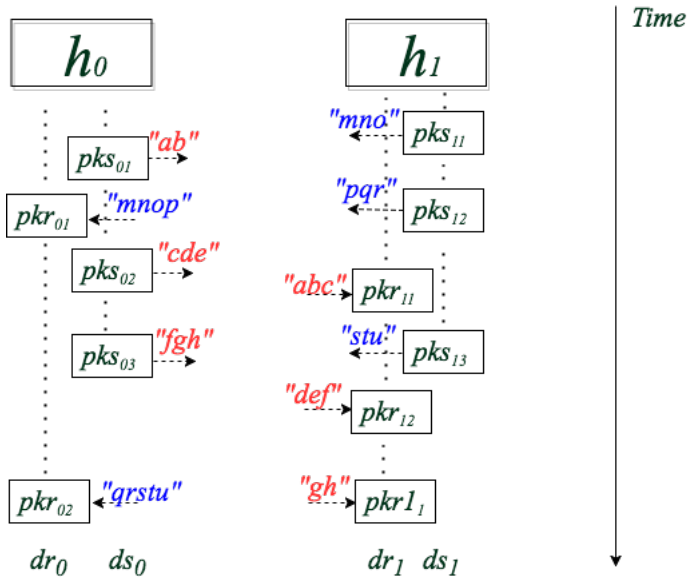

Figure 3.1: Example of reliable communication

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.

By concatenating the payload of the sent packets in ds_0 and the received packets in dr_1, I notice that the concatenations are equal:

$pl(pks_{01}) \cdot pl(pks_{02}) \cdot pl(pks_{03}) = pl(pkr_{11}) \cdot pl(pkr_{12}) \cdot pl(pkr_{13}) =$ "$abcdefgh$"

In the other direction, the concatenations of the payload of the sent packets in ds_0 and the concatenation of the received packets in dr_1 are equal:

$pl(pks_{11}) \cdot pl(pks_{12}) \cdot pl(pks_{13}) = pl(pkr_{01}) \cdot pl(pkr_{02}) =$ "$mnopqrstu$".

So this communication satisfy the content preservation.

From the Figure 3.1, it is obvious that 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 $pl(pks_{01}) \cdot pl(pks_{02}) \cdot pl(pks_{03}) =$ "$abcdefgh$"

prove at any given time during this communication, the recipient only received at most the data that has been sent. So this communication satisfy the timing preservation.

Figure3.2 is an example of an unreliable communication.

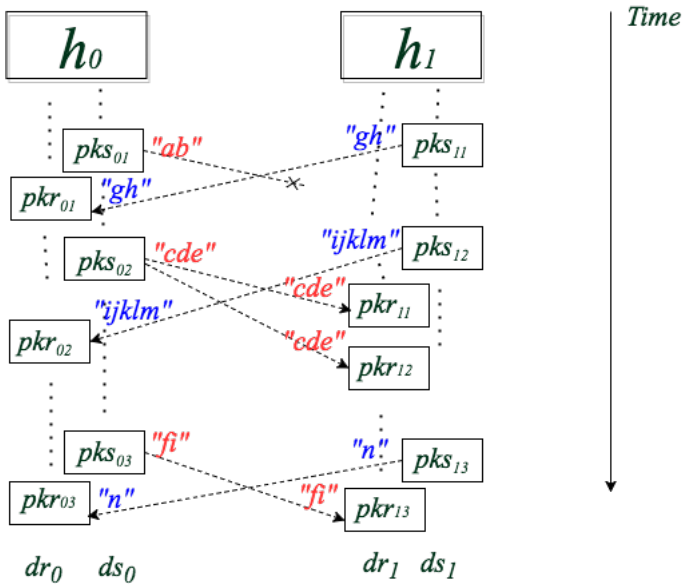

Figure 3.2: Example of unreliable communication

In this example, the content preservation of the unreliable communication are satisfied since each received packet has a matched sent packet on the other side:

$pkr_{11} = pks_{02} =$ "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".

The timing preservation of the unreliable communication are satisfied since the match of the received packets (the sent packets) had been sent before the received packets are received.

$time(pkr_{11}) > time(pks_{02})$;
$time(pkr_{12}) > time(pks_{02})$;
$time(pkr_{13}) > time(pks_{03})$;
$time(pkr_{01}) > time(pks_{11})$;
$time(pkr_{02}) > time(pks_{12})$;
$time(pkr_{03}) > time(pks_{13})$;

Chapter 4

Communication Analysis

I defined a message transferring communication between two programs in Chapter 3. The goal of this research is to develop a method to identify the communications from a 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 I formalize the dual_trace. For all the traces that comply with this abstract dual_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 functions descriptor. These two sequences of events of these two traces will then flow into the stream extraction component separately. In each event sequence, the events might be triggered 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 and see if it satisfy the communication content 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.

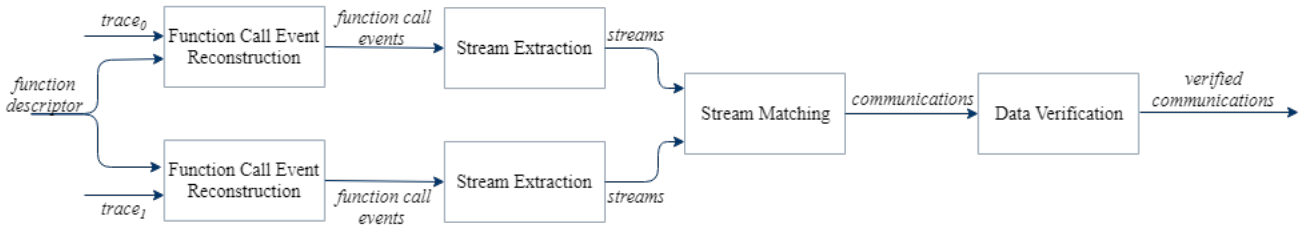

Figure 4.1: Process of the communication analysis through a dual_trace

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.

A dual_trace is formalized as :

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

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

An execution trace consists of a sequence of instruction lines and can be defined as:

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

Each instruction line contains the executed instruction, the changed memory, the changed registers and the execution information and can be defined as 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, and system call exit, $syscallInfo =< exeName, funcName >$ 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 with the definition of a trace.

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

Figure 4.2: An example trace

4.2 Functions Descriptors

There could 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, such as TCP, UDP, and Named Pipe. To be able to identify and reconstruct the function calls, I define a functions 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 function) of a given function call, and $outparamdesc$ is the output parameter descriptions similar to the input parameter descriptions.

Table 4.1 is an example of a function description. In this example, the function name is $ReadFile$, it is a function for data receiving, so its 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 a 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-level execution trace and defined the functions descriptor of a communication method. The functions 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 an algorithm to reconstruct the function call events from an assembly-level execution trace.

With the functions 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 through the whole execution trace, the algorithm outputs a sequence of function call events of length m. This sequence of events can be defined as etr:

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

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 is inherited from the function description and can be one of the four types: $open$, $send$, $receive$ and $close$.

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

Algorithm 1 presents the pseudocode 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 analyze each of them. Since there are usually a small number of functions of interest for a communication method compared to the number of instruction lines in the execution trace, the time complexity of this algorithm is $O(N)$ and N is the number of instruction lines in the trace.

Algorithm 1: Function Event Reconstruction Algorithm

```
/* trace is the assembly-level execution trace with a sequence of instruction lines:
   (l_1, l_2, ..., l_n), cdesc is the functions 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             i ← i + 1
14             while i ≤ n do
15                 l ← trace[i]
16                 i ← i + 1
17                 Execute the instruction of l
18                 if l is a exit of the function described by fdes then
19                     Get the output parameters from the memory state and append them to ev.outparams
20                     Break the inner while loop
21             etr.append(ev)
22             Break the For loop
23 return etr
```

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:{}}
```

4.4 Channel Open Mechanisms

The channel open mechanism affects the stream extraction and stream matching strategy. So I discuss them before presenting those algorithms. The channel open mechanism of a 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 them 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 an 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 endpoints is closed, its handle can be reused for other newly 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 the different communication methods open their channels for communication.

4.4.1 Named Pipe Channel Open Mechanisms

In the Named Pipe communication method, a named pipe server is responsible for the creation of the pipe. The creation of a named pipe returns the file handle of that pipe. So on the server side, the

identification of the stream needs to identify the pipe creation function call. Clients can connect to the pipe with the pipe name after it is created. So, on the client side, the identification of the stream is to identify the pipe connection function call. The handle returned by the pipe creation and connection function calls will be used later when data is being sent to or received from 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. [15] Figure 4.3 exemplifies the channel set up process for a Named Pipe communication in Windows.

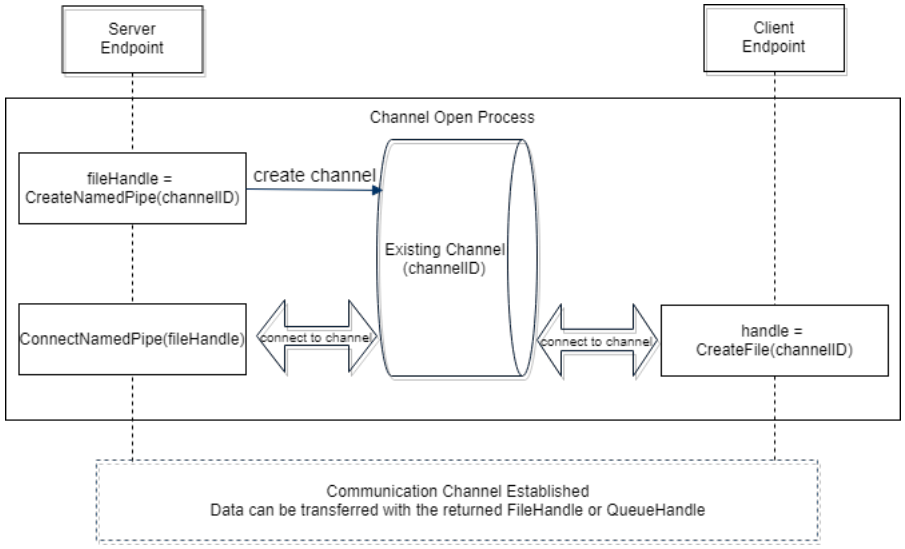

Figure 4.3: Channel open process for a named pipe in Windows

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 accessing 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 identifier of the channel is the input for the open queue function. [14] Figure 4.4 exemplifies the channel set up process for a Message Queue communication in Windows.

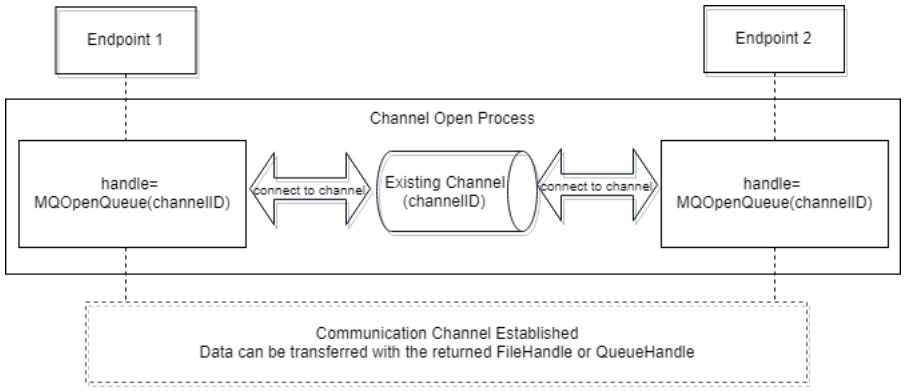

Figure 4.4: Channel open process for a message queue in Windows

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 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 calls 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, and the second one is used for real data transfer. The server's address and port are considered to be the identifier of the channel [16]. Figure 4.5 exemplifies the channel open process for TCP and UDP in Windows.

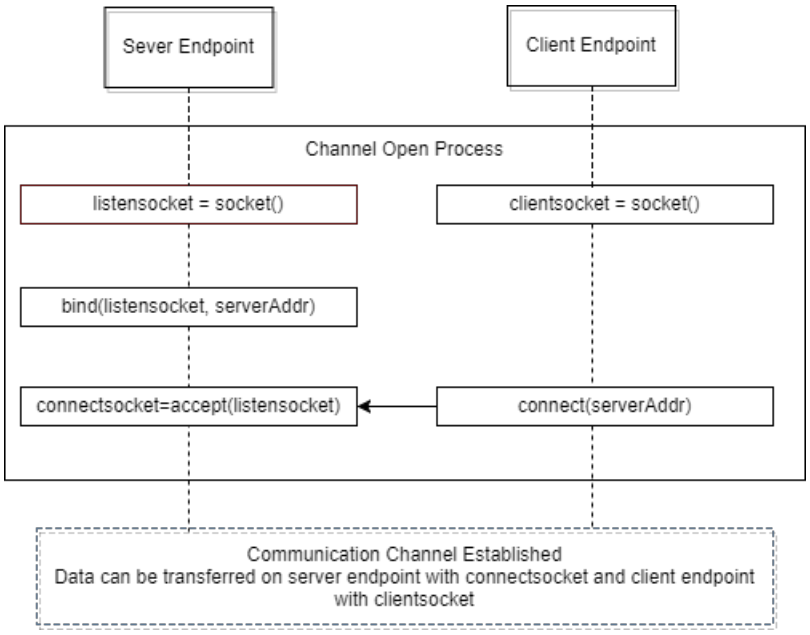

Figure 4.5: Channel open model for TCP and UDP in Windows

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 separate 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 substreams which are called open stream, send stream, receive stream, and close stream. There will be only one type of event in each of these streams. The reason to divide a stream into sub streams is that the later stream matching only ~~need~~ needs the information extracted from the open stream and the data verification only ~~need~~ needs the data extracted from the send and receive streams. So separating them will simplify the later processes. 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, sx is defined as:

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

The event numbering of in this sub stream is different from the original sequence of event. For example, ev_1 in sx and ev_1 in etr might be different events.

The stream extraction algorithms are designed to separate the streams from a sequence of function call events. In these algorithms, 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 for Named Pipe and Message Queue, while the other for TCP and UDP.

4.5.0.1 ~~Stream Extraction Algorithm for Named Pipe and Message Queue~~

4.5.1 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 endpoint handle is found.

The same handle may be reused by another endpoint once it is closed by the channel close function call. 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 temporary 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
```

4.5.1.1 ~~Stream Extraction Algorithm for TCP and UDP~~

4.5.2 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 the socket create function in both client and server. On the server side, this created socket is only used for listening to the client's connection. The listening is accomplished by calling the accept function. 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. 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.

Similar to the algorithm for Named Pipe and Message Queue, the reuse 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 check for 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
1  str ← ∅;
   /* a temporary stream set for all open streams                                     */
2  tempstreams ← ∅;
3  for ev ∈ etr do
4      if ev.funN = socket then
5          h ← the handle in ev.outparams;
6          if tempstreams[h] not exist then
7              tempstreams[h] ← a new s                    // a new stream;
8              tempstreams[h].handle ← h;
9              tempstreams[h].so.append(ev);
10     else if ev.funN = bind or ev.funN = connect then
11         h ← the handle in ev.inparams;
12         if tempstreams[h] exist then
13             tempstreams[h].channelId ← address and port parameter in ev.inparams;
14             tempstreams[h].so.append(ev);
15     else if ev.funN = accept then
16         h ← the handle in ev.inparams;              // the handle of parent stream;
17         hc ← the handle in ev.outparams;            // the handle of child stream;
18         if tempstreams[h] exist then
19             if tempstreams[hc] not exist then
20                 tempstreams[hc] ← a new s;          // a new stream for the child;
21             tempstreams[hc].handle ← hc;
22             tempstreams[hc].channelId ← tempstreams[h].channelId;
23             tempstreams[hc].so.append(tempstreams[h]);      // append parent's events;
24             tempstreams[hc].so.append(ev);          // append the current event;
25     else if ev.type = send then
26         h ← the handle in ev.inparams;
27         if tempstreams[h] exist then
28             tempstreams[h].ss.append(ev);
29     else if ev.type = receive then
30         h ← the handle in ev.inparams;
31         if tempstreams[h] exist then
32             tempstreams[h].sr.append(ev);
33     else if ev.type = close then
34         h ← the handle identifier from ev.paras;
35         if tempstreams[h] exist then
36             tempstreams[h].sc.append(ev);
37             str.append(tempstreams[h]);
38             remove tempstreams[h] from tempstreams;
39     else
40         unknown event type or name error;
41 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 from 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, the 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_1 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 when the same channel is reused by the different endpoints in the same program. 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 in this case, there will be two matchings.

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

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

4.7 Data 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 verifies 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 model consist of content preservation and timing preservation. 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 in a stream but not crossing two streams. So 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 verification algorithms are two preliminary matched streams s_0 and s_1. The output is a boolean indicating if the streams satisfy 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 needs to be verified and for Message Queue and UDP, the unreliable communication preservation needs 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 verification algorithm. The data transfer properties and scenarios are summarized from the perspectives of the how the protocol normally behave. So the output communications after data verification are those aligning with the those normal communication properties. By comparing the output extracted streams and the communications, the software security engineer might be able to detect the malicious streams and further detect the problems from the programs.

4.7.1 Data Verification Algorithm for Named Pipe

Named Pipe provides First In First Out (FIFO) communication mechanism for inter-process communication. It can be a one-way or a duplex pipe [15]. 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 happens 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.

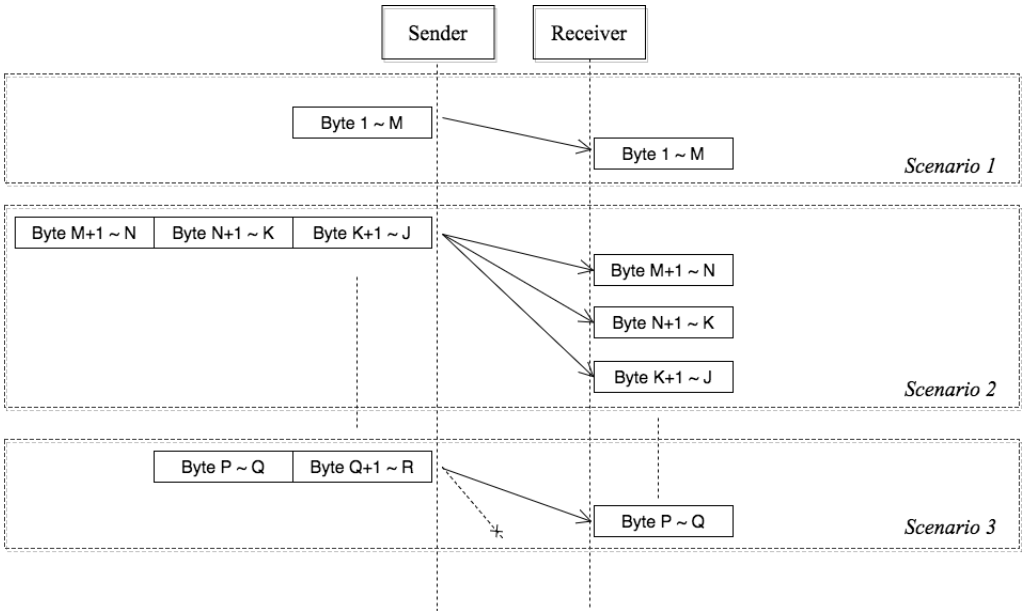

Figure 4.6: Data transfer scenarios for Named Pipe

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 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

1 **return** $satisfied$

2 $send_0 \leftarrow$ concatenation of the payload of send function call events in $s_0.ss$;

3 $send_1 \leftarrow$ concatenation of the payload of send function call events in $s_1.ss$;

4 $receive_0 \leftarrow$ concatenation of the payload of receive function call events in $s_0.sr$;

5 $receive_1 \leftarrow$ concatenation of the payload of receive function call events in $s_1.sr$;

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

4.7.2 Data 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.

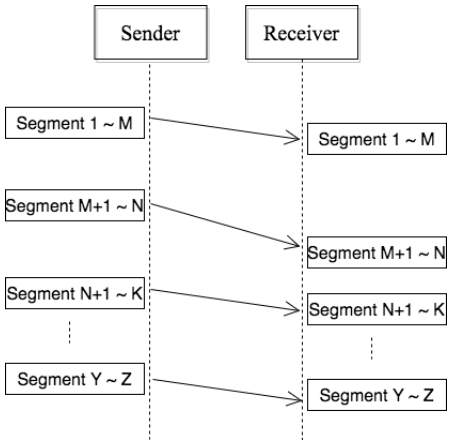

Figure 4.7: Data transfer scenarios for TCP

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. 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. 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 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
return satisfied
/* There is a chance that the trace capturing end before the channel is closed          */
for i ∈ 0..min(s_0.ss.size, s_1.sr.size) do
    if s_0.ss[i].payload != s_1.sr[i].payload then
        return False;
for i ∈ 0..min(s_1.ss.size, s_0.sr.size) do
    if s_1.ss[i].payload != s_0.sr[i].payload then
        return False;
return True;
```

4.7.3 Data 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 [17]. 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 are 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.

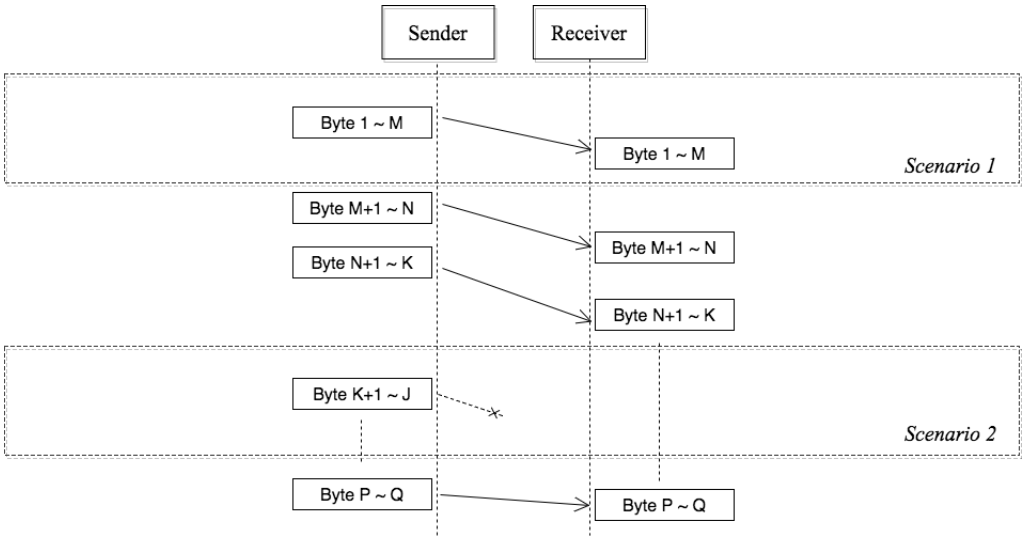

Figure 4.8: Data transfer scenarios for Message Queue

To verify the content preservation of the unreliable communication, for each received packet, Algorithm 7 tries to find the matched sent packet in the other stream. If any of the received packets cannot be matched, the content preservation is not satisfied. Since the sent packets are received in order, the search 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 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 cannot 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 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.

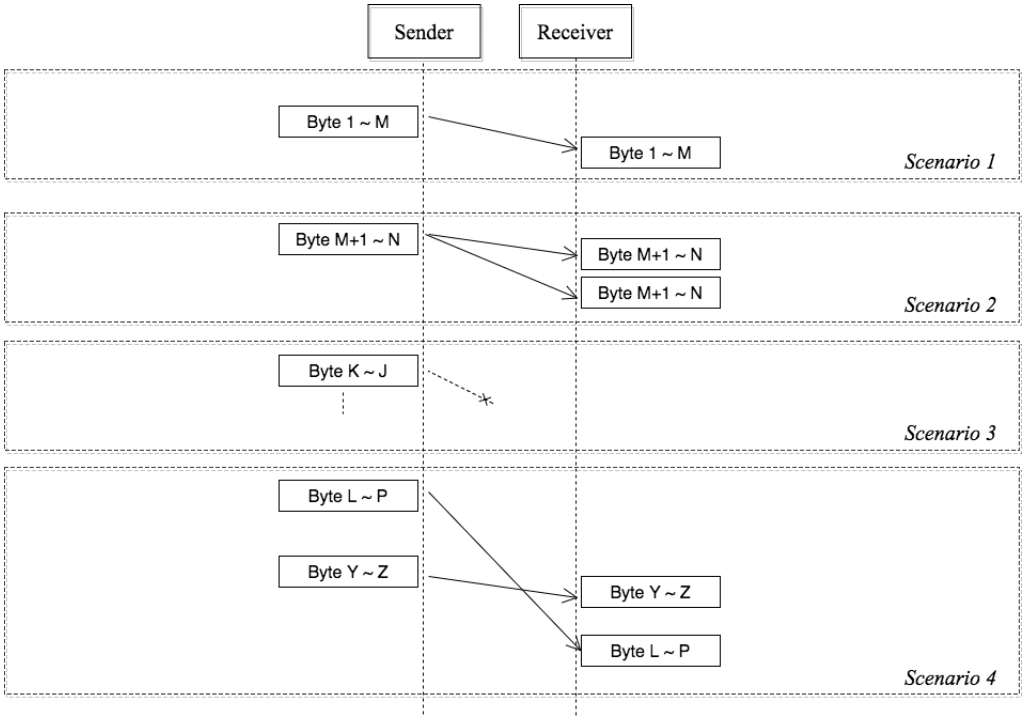

Figure 4.9: Data transfer scenarios for UDP

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 cannot 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
1  return satisfied
2  if s_0.ss.size < s_1.sr.size Or s_1.ss.size < s_0.sr.size then
3      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                                                                        */
4  for i ∈ 0..s_1.sr.size do
5      matchFlag = False;
6      for j ∈ 0..s_0.ss.size do
7          if s_0[j].ss.payload = s_1[i].sr.payload then
8              matchFlag = True;
9              delete the packet from sends_0 break the inner For loop;
       /* This received packet cannot be matched by any sent packet                        */
10     if matchFlag = False then
11         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                                                                        */
12 for i ∈ 0..s_0.sr.size do
13     matchFlag = False;
14     for j ∈ 0..s_1.ss.size do
15         if s_1[j].ss.payload = s_0[i].sr.payload then
16             matchFlag = True;
17             delete the packet from s_0.ss break the inner For loop;
18     if matchFlag = False then
19         return False;
20 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 cannot be excluded even with the content verification

The reason that the timing preservation cannot be verified is lacking timing information across the two traces.

For the second limitation, the main reason is that the data transmitted in two communications could 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. In order to reduce this type of error, I developed the data verification strategy and algorithms. However, in some cases, the transferred data of two communications can be identical or very similar. Even with content verification, the mismatched streams still cannot be 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 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.

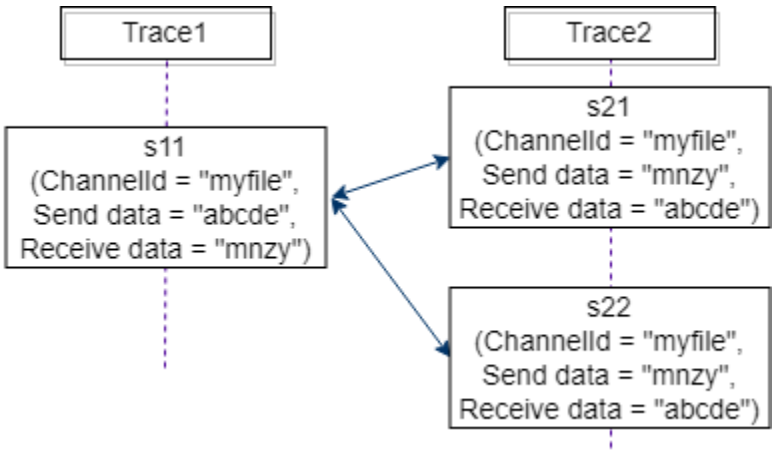

Figure 4.10: An ineffective stream matching scenario

4.8 Discussion of Security Analysis Challenge

Malware is a serious problem for all system nowadays. The intrusion of malware is generally an individual behavior, malwares attack the vulnerability of one or several specific systems, then intrudes into the system, destroys and controls it to obtains private information. Some malware can even be spread like a virus through the Internet, invading other systems with the same vulnerability. In general, malware is targeted at system-level intrusions, and there is very little damage to common

application software unless there are obvious vulnerabilities that can be exploited for systematic attacks. [13]

On the other hand, network attacks often occur on the network layer. Since the internet has be an extremely huge network in the world, the attack could be initiated at any network node. If there are no proper security measure, your data might be attacked during the transmission. There are many types of network attack, some are passive like monitoring information, some are active like corrupting and destroying the data or network itself. [10]

There is not a system that is absolutely secure from attackers. The running system's behaviour can change dramatically due to all of these attacks. The goal of the communication analysis is simply monitoring programs in terms of data transmission. Perhaps the communication is modified by the attacks mention above. This places a huge challenge for communication analysis. For example, the identification of a modified communication would be an extremely difficult mission.

However, even though there might be some cases that the communication can not be effectively identified, this work will still be valuable to the analyst, as it will show that the data sent was not received (at least the same data was not received) by the other side and might mean that the running environment is compromised.

Chapter 5

Dual_trace Communication Analysis Prototype In Atlantis

In this chapter, I present the design of the prototype of communication analysis from the dual_trace. This prototype is implemented in Atlantis. Atlantis is an assembly-level 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 functions 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, and 4) a view for presenting the extracted streams and the identified communications from the dual_trace.

5.1 Use Cases

In this section, two 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 needs to perform the below operations in sequence:

- Open two traces in a parallel view (as part of this prototype)
- Import the dynamic-linked library files for each trace (as part of the 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 results 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. The goal of this use case is to analyze two execution traces in a dual_trace and output the streams of each trace. The use case shown in Table 5.2 is for identifying the concerned communication from a dual_trace.

Table 5.1: Use case 1: extract streams from a dual_trace

Name	Analysis streams of a communication method from the Dual_trace
Description	A user capture two assembly-level 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-level 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 functions descriptors for the communication methods of interest in a Json format setting file 2. The user opens one of the trace in Atlantis 3. The user opens the other trace as the dual_trace of the first one 4. The two opened traces are presented in the parallel view 5. The user loads the related .dll files for both opened traces 6. The user selects the operation "Stream Extraction" in the "Dual_Trace Tool" menu. 7. Atlantis prompts a dialog window giving the user all the communication methods in the functions descriptor setting file as options 8. The user selects the communication methods that they want to analyze and click the "OK" bottom 9. Atlantis extracts the streams for both traces and lists the results in the "Communication view" 10. The user expands the result in the "Communication view" 11. The user selects one function call event in a stream and double clicks the entry 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 captures two assembly-level execution traces of two interacting programs and needs to analysis them by identifying all communications of the dual_trace and inspects the extraction results
Actor	A software security engineer
Precondition	The user has two assembly-level 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 functions descriptors for the communication methods of interest in a Json format setting file 2. The user opens one of the trace in Atlantis 3. The user opens the other trace as the dual_trace of the first one 4. The two opened traces are presented in the parallel view 5. The user loads the related .dll files for both opened traces 6. The user selects the operation "Communication Identification" in the "Dual_Trace Tool" menu 7. Atlantis prompts a dialog window giving the user all the communication methods in the functions descriptor setting file as options 8. The user selects the communication methods that they want to analyze and click the "OK" bottom 9. Atlantis identifies the communications of the dual_trace and lists the results in "Communication view" 10. The user expands the result in the "Communication view" 11. The user selects one function call event in a stream and double clicks the entry 12. Atlantis shows the corresponding instruction line in the trace and synchronize all other views

5.2 Declaring of the Functions Descriptors

In Section 4.2, I described how to develop a functions descriptor for each communication method. Each function description consists of 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. The communication analysis approach depicted in Chapter 4 can identify the communications described by the functions descriptor from the dual_trace.

However, the functions descriptor for a communication method can be different depending on the implementation of the communication method in a program. Rather than hard coding the functions descriptors for the communication methods, this prototype loads the functions 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 to match the communication methods of interest. The default template example can be found in Appendix B.

The functions 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 functions descriptor configuration file will be presented to them. They can select one or more communication methods to be analyzed.

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

5.2.1 Communication Methods' Implementation in Windows

Learning the implementation of a communication is necessary to obtain the functions 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 functions descriptors of these methods.

The Windows API set is very complex. Moreover, multiple solutions are provided to fulfil a communication method. It is not feasible within the scope of this thesis to enumerate all solutions for each communication method. I only investigated the most basic usage provided in the Windows documentation. For each communication method, a functions 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 and Window 7. The provided functions descriptor of a communication method should only be considered as an example for that communication method. With this understanding, it should be fairly easy to obtain the functions 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 functions descriptor is generalizable to other operating systems given some the effort to understand the APIs of those operating systems.

5.2.1.1 Windows Calling Convention

For this research, it is important to know the 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, 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 as a case study for this research. The Microsoft* x64 calling convention is listed in Appendix A for reference.

5.2.1.2 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 [15]. 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 functions 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 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 shows the functions descriptor for synchronous mode while Table 5.4 is the functions descriptor asynchronous mode of Named pipe.

Table 5.3: Functions 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: Functions 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

5.2.1.3 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 finishes, the parameter in RCX of this function call is a structure consisting of the handle as an input parameter and the overlap structure as an output parameter. Table 5.5 shows the functions descriptor for synchronous mode while Table 5.6 is the functions descriptor for the asynchronous mode without callback. I did not exemplify the case with a callback function, since the specific callback function and its parameters need to be known for developing the functions descriptor.

Table 5.5: Functions 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: Functions 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

5.2.1.4 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 shows the functions descriptor for UDP or TCP communication.

Table 5.7: Functions descriptor 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 Eclipse plug-in development. The details of the implementation can be found in Appendix C. 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.

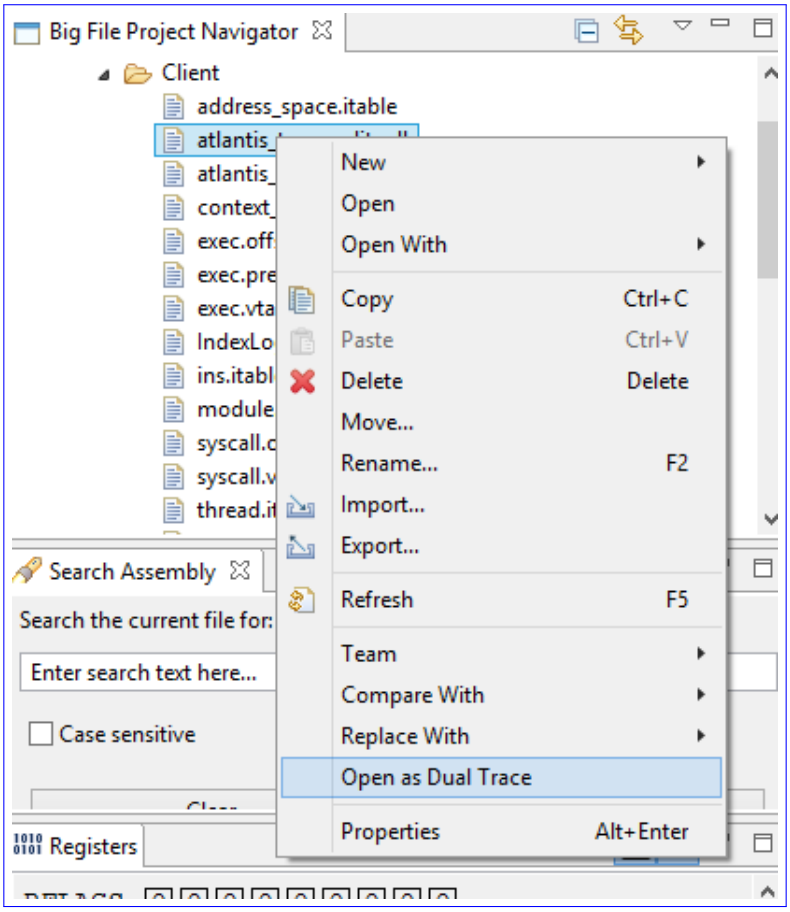

Figure 5.1: Menu item for opening dual_trace

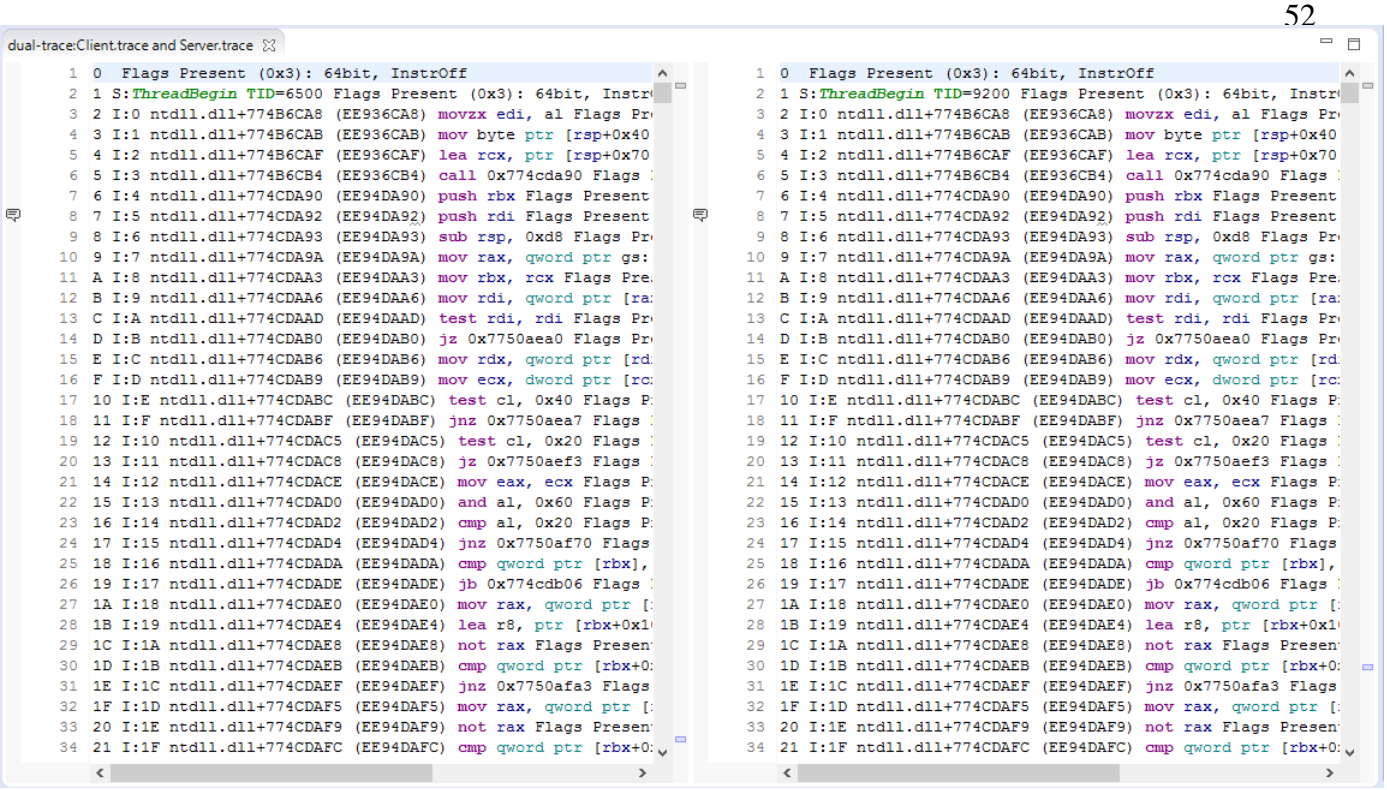

Figure 5.2: Parallel trace view

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.

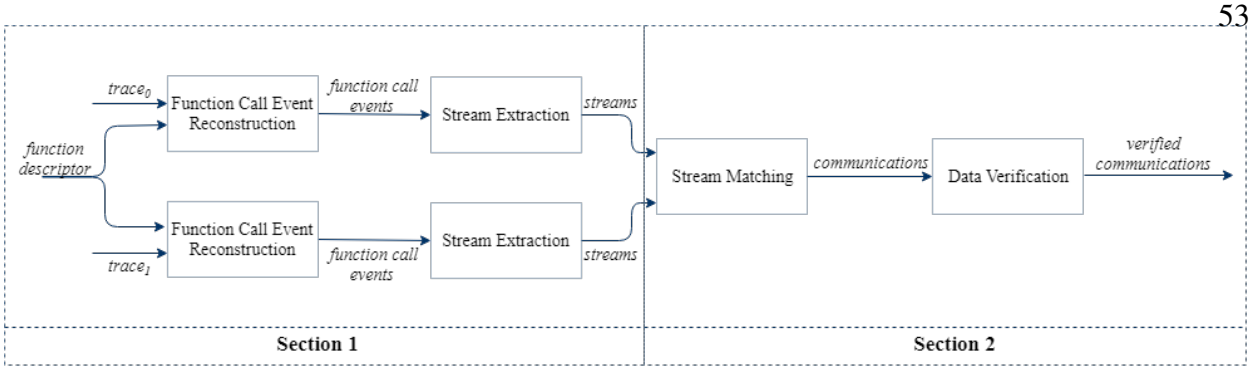

Figure 5.3: Process of the communication analysis from a dual_trace separated in two sections

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. Instead of having the function name in $syscallInfo$, the current traces only provide the offset of that function in the corresponding executable file. So, 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' names can be inspected by searching for the symbolic name in the executable binary or any DLLs used by the program at the time when it is traced. Figure 5.4 shows an example of the trace from DRDC. In this example, line 1504 is a system call entry with the system call information as $syscallInfo =< kernel32.dll, 0x10D10 >$. Figure 5.5 is the information decoded by the "function inspect" feature of Atlantis from the executable file kernel32.dll of the system where the trace shown in Figure 5.4 was captured. 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 to the one align to the dual_trace formalization.

Line	Instruction	Memory Changes	Regitster Changes	Execution Type	System Call Info
...	...	...	...	...	...
1499	lea rcx, ptr [rip+0x3cfe0]		ECX F8A0AAA8	Instruction	
1500	xor r9d, r9d			Instruction	
1501	mov edx, 0xc0000000		EDX C0000000	Instruction	
1502	mov dword ptr [rsp+0x20], r8d	00000000001DF490 00000003 00000000	RSP 001DF470	Instruction	
1503	call qword ptr [rip+0x3a97d]	00000000001DF470 000007FE F89CDADB	RSP 001DF468	Instruction	
1504	mov qword ptr [rsp+0x8], rbx	00000000001DF470 00000000 00000001	EBX 00000001	System Call Entry	kernel32.dll+0x10d10
1505	mov qword ptr [rsp+0x10], rbp	00000000001DF478 00000000 00000000	EBP 00000000 ESP 001DF468	Instruction	
...	...	...	...	...	...
2057	add rsp, 0x20		ESP 001DF2A0	Instruction	
2058	pop rbx		RBX 001DF3E8 RSP 001DF2A8	Instruction	
2059	ret		RSP 001DF2B0	System Call Exit	kernel32.dll+0x10d10
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	CreateloCompletionPort
0x5c1e0	CreateJobObjectA

Figure 5.5: Information from kernel32.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, The "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 is run separately for both traces in the dual_trace, the traces are transformed into the proper format for communication analysis. Figure 5.6 shows this new menu in Atlantis. When the user performs the "Stream Extraction" or "Communication Identification" operations, 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 selects and confirms the communication methods.

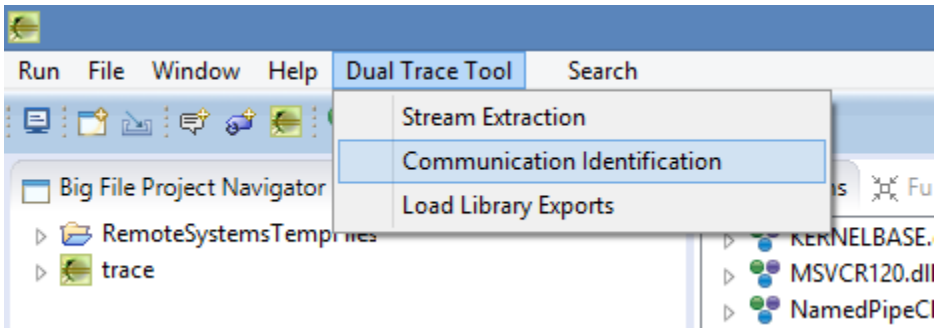

Figure 5.6: Dual_trace tool menu

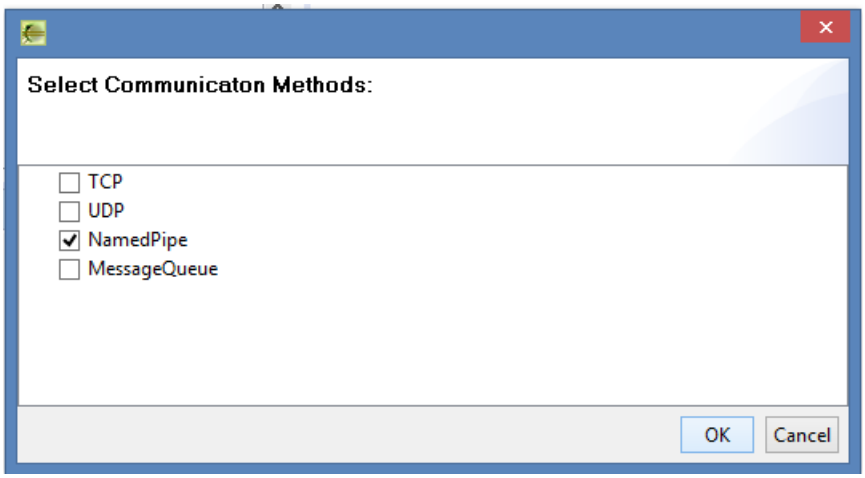

Figure 5.7: Prompt dialog for communication selection

5.5 View of Extracted Streams and Identified Communications

I designed a new view named "Communication View" 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 results contain 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

runs 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~~ 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.

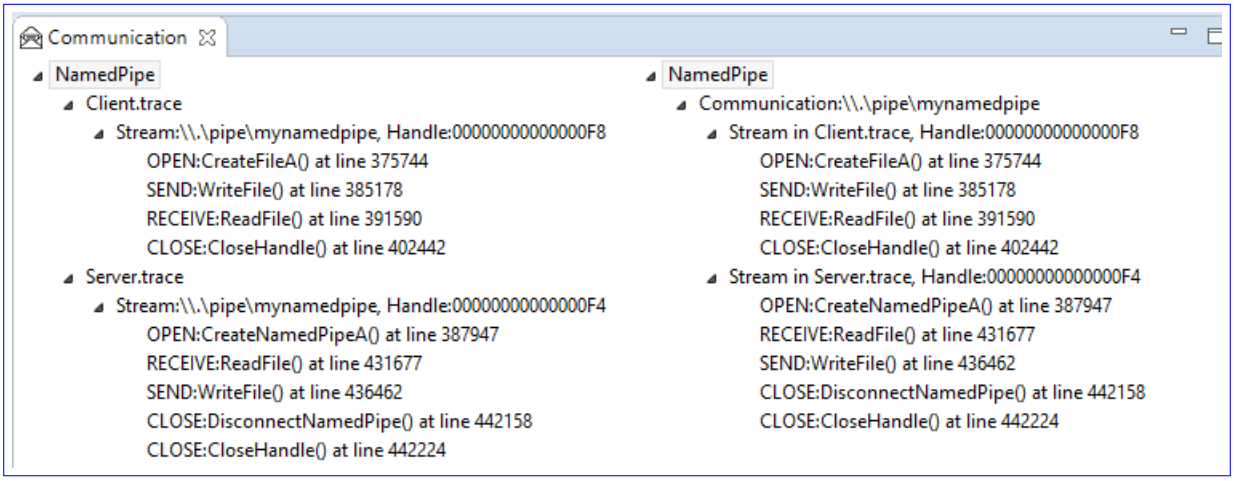

Figure 5.8: Communication view for results

Atlantis is an analysis environment that has various views to provide user access to different information from a trace, such as the memory and register state of the current instruction line. Moreover, these views synchronize automatically with the trace view. This functionality and information also benefits 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 view 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 a method to easily access these two instruction lines, from the event entries, this implementation provides 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.

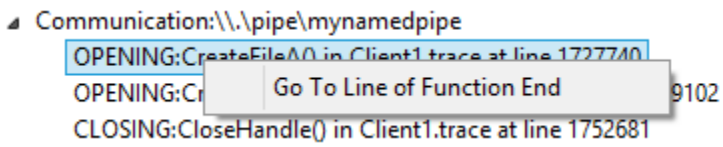

Figure 5.9: Right click menu on event entry

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.

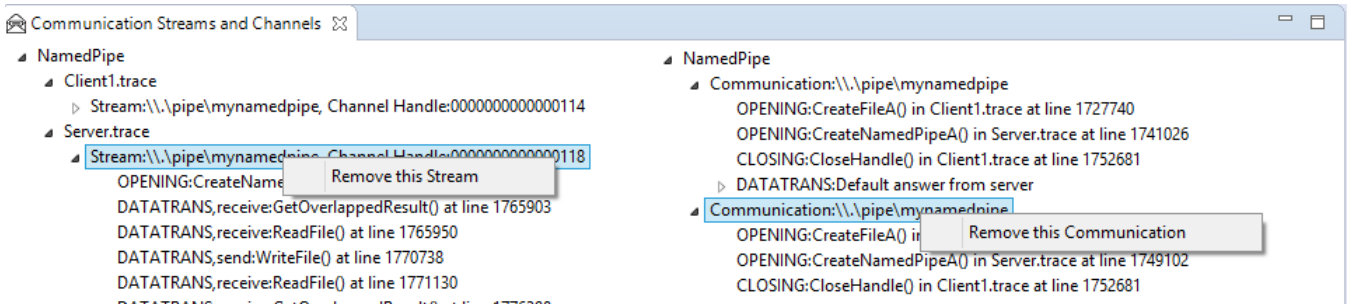

Figure 5.10: Right click menu on event entry

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.

The selection of 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. Thus both of the conducted experiments used the Named Pipe communication method.

Among these two experiments, the first one was provided directly by our research partner DRDC with their initial requirement, while I designed the second one. 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 results by navigating the events in the "Communication view" and check if they match the sequence diagrams of each experiment

In the following two sections, I describe the design of the experiments first and then present the results 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.

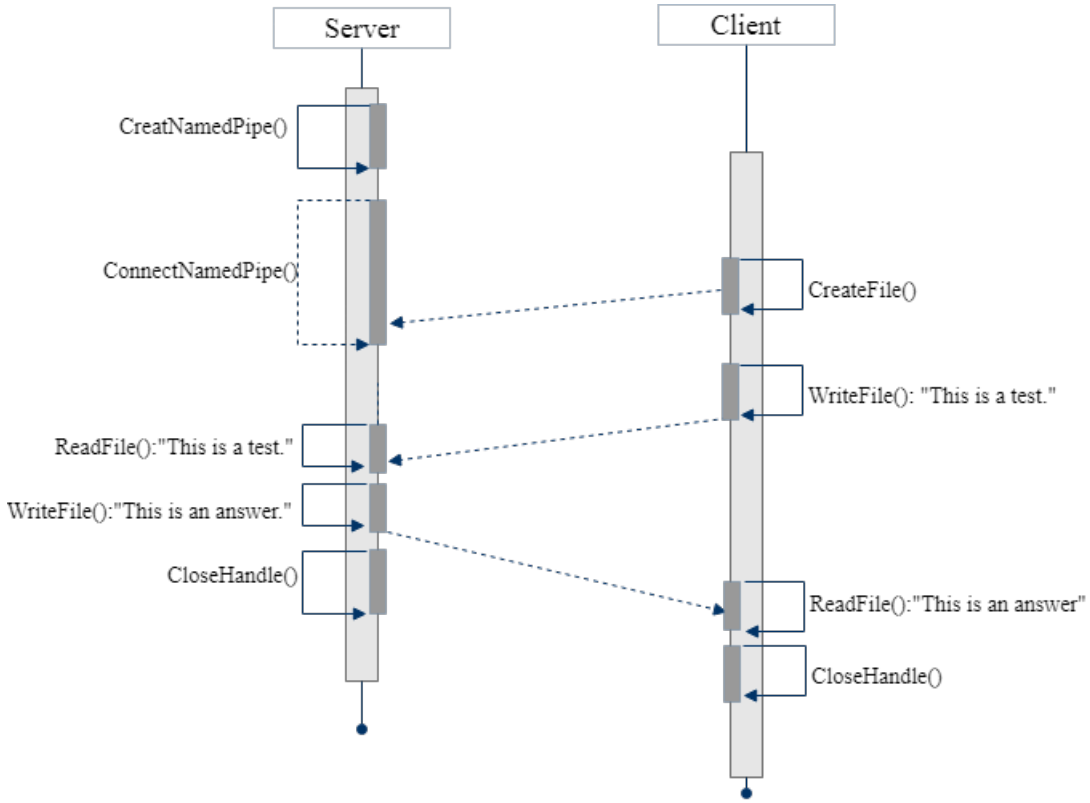

Figure 6.1: Sequence diagram of experiment 1

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 functions descriptor for the analysis of $dual\_trace\_1$ is shown in Table 6.1.

Table 6.1: Functions 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 Results Walk Through

$Client.trace$ has 412,717 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 a stream was extracted, which consists of all these four function call events.

$Server.trace$ has 461,817 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, a stream was extracted, which consists of all of these five function call events.

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

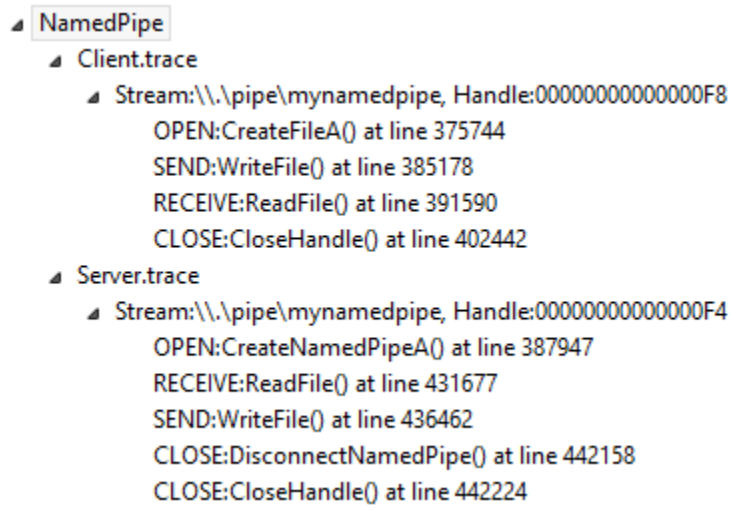

Figure 6.2: Extracted streams of $dual\_trace\_1$

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 the stream 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 "$This\ is\ a\ test.$". 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 streams 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 the "Communication view" in Atlantis as shown in Figure 6.3.

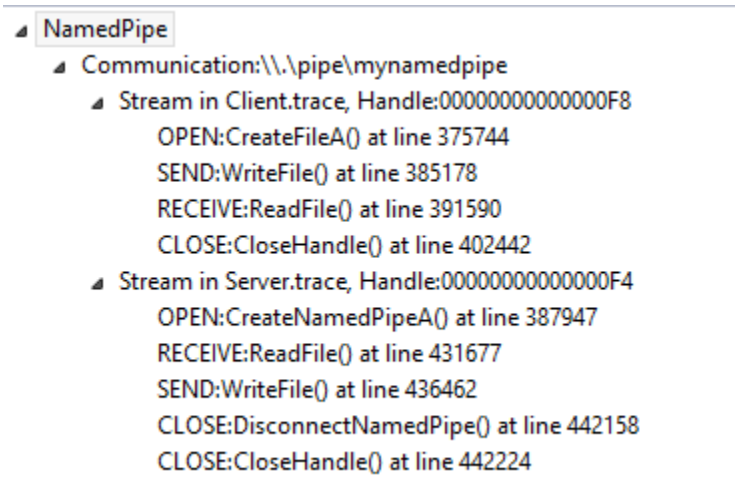

Figure 6.3: Identified communication of $dual\_trace\_1$

After seeing 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 in Figures **??** and **??**.

Figure 6.4 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 6.5 shows when I selected "Go To Line of Function End" from the right click menu on the $ReadFile$ function call event of $Server.trace$, that 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 *"$This\ is\ a\ test$."* is transmitted from the client to the server.

Figure 6.6 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 6.7 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

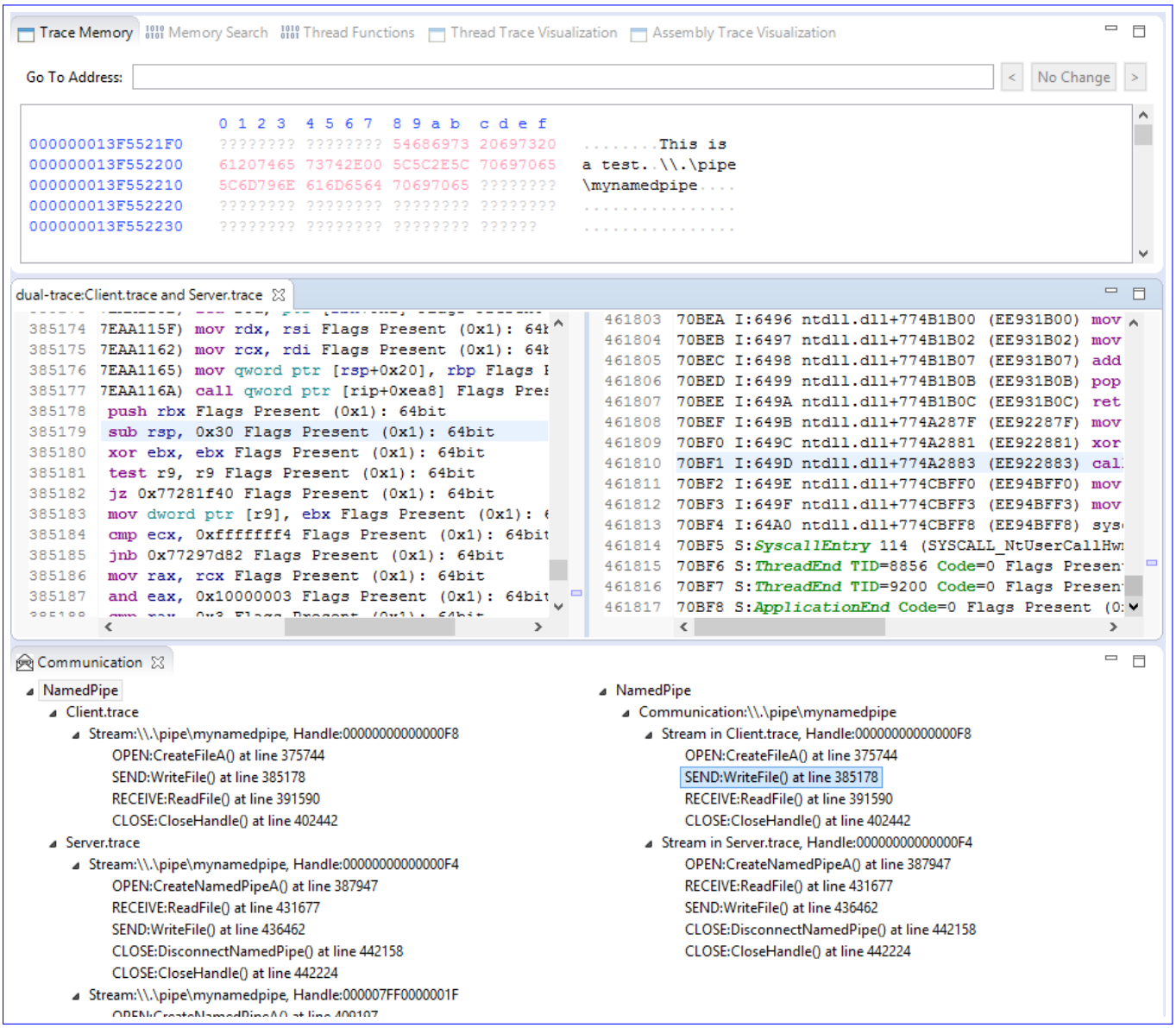

Figure 6.4: Client send event navigation for the message *"This is a test."*

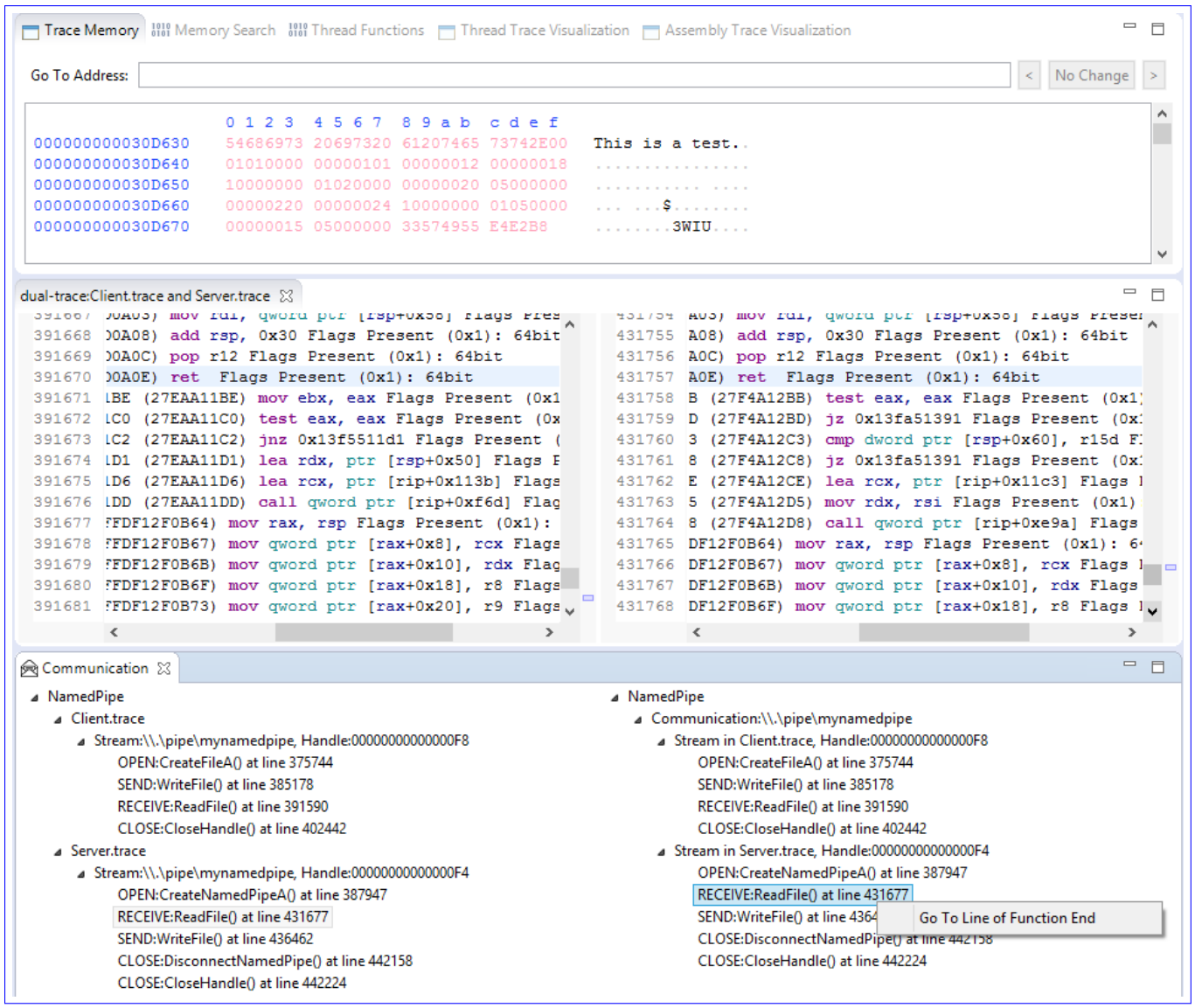

Figure 6.5: Server receive event navigation for the message *"This is a test."*
~~Navigation results for the transmitted message *"This is a test."*~~

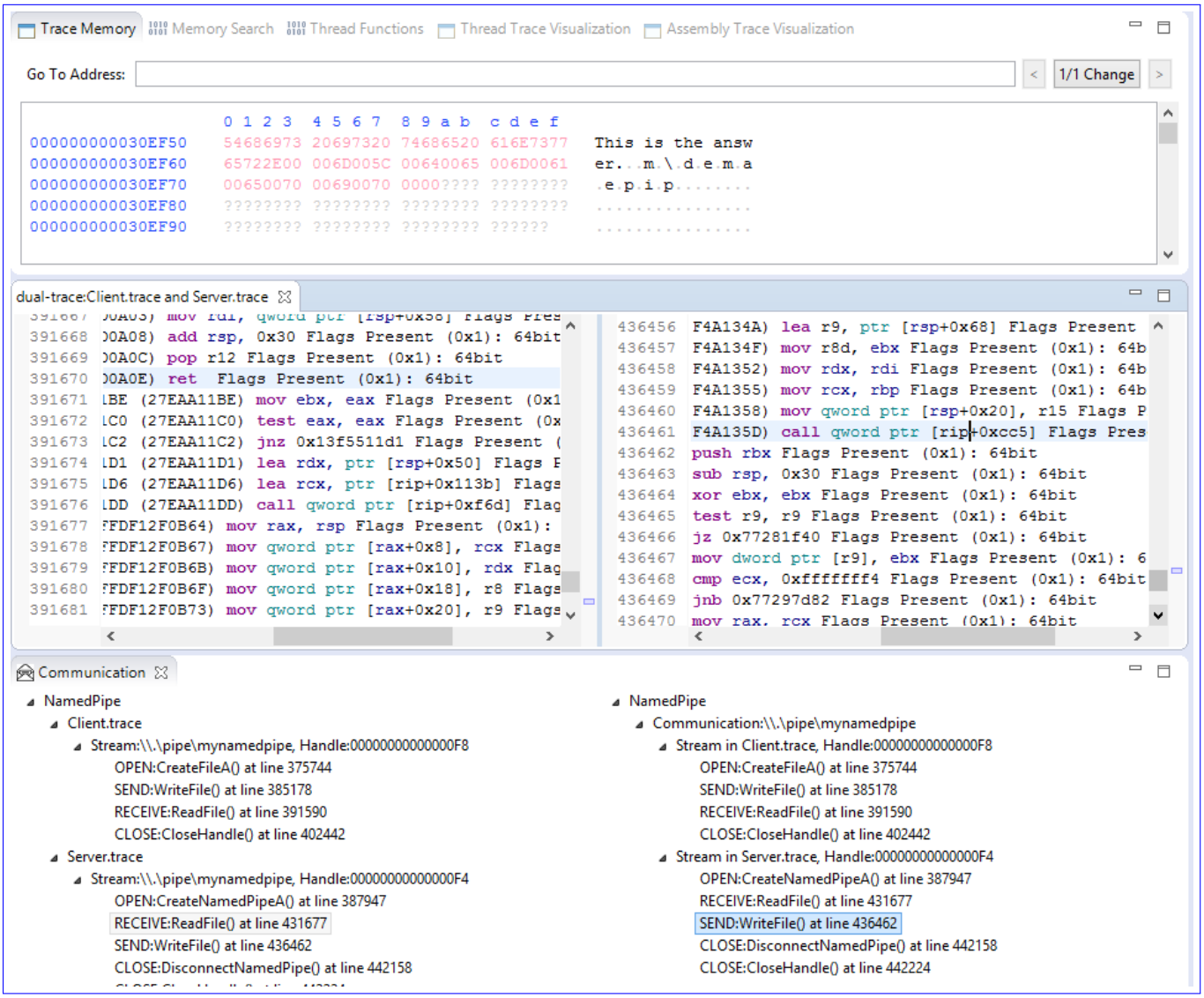

Figure 6.6: Server send event navigation for the message *"This is the answer."*

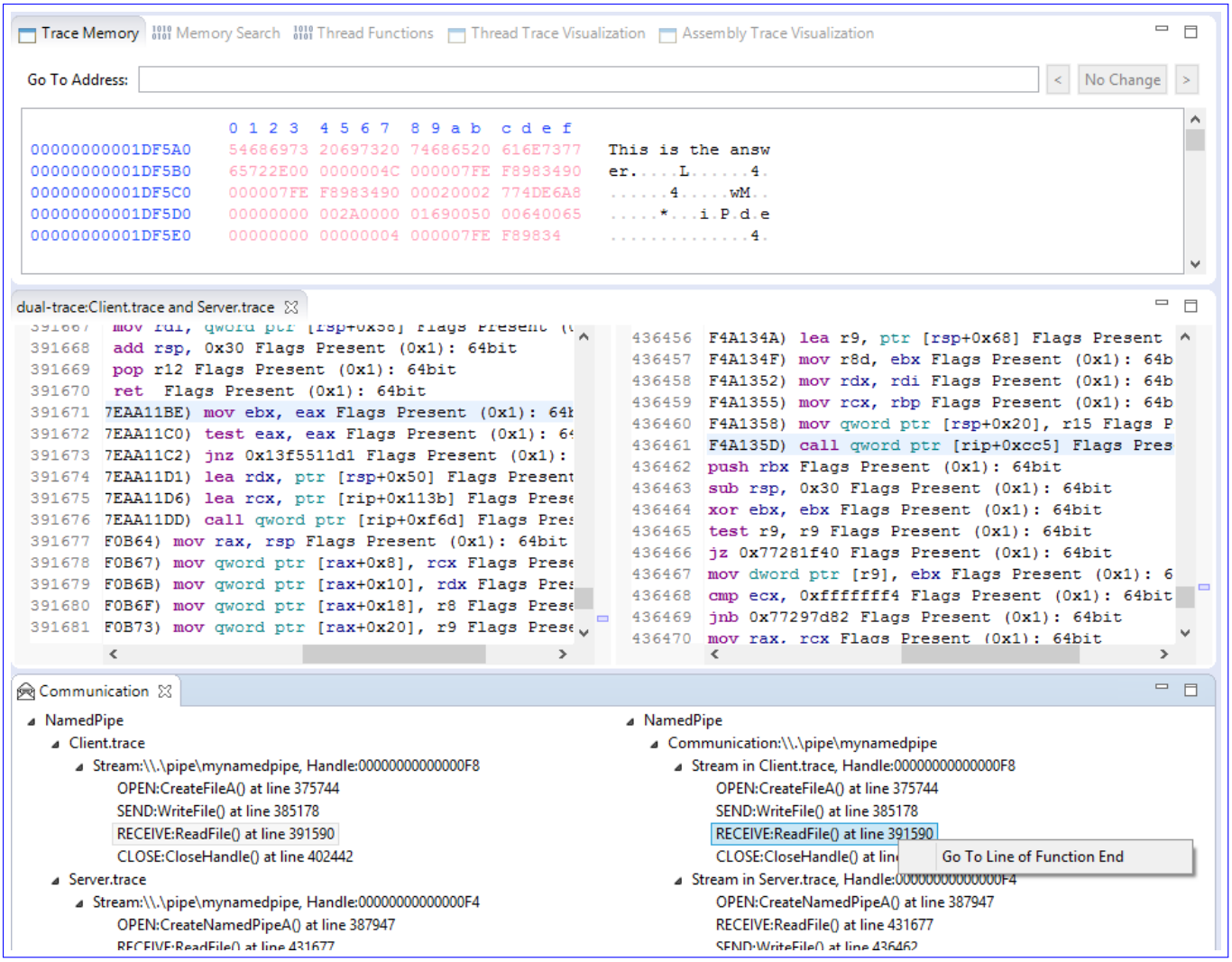

Figure 6.7: Client receive event navigation for the message *"This is the answer."*
~~Navigation results for the transmitted message *"This is the answer."*~~

to the memory address $0x1DF5A0$, which is the address for the receive buffer of the message "$This\ is\ the\ answer$.".

These two figures perfectly show how the message "$This\ is\ the\ answer$." is transmitted from the server to the client.

6.2 Experiment 2

6.2.1 Experiment Design

In the second experiment, one server program and one client program are written in C++. In the server program, four named pipes were created and could be connected by up to four clients at a time. The named pipe client program was run two times in sequence to act as two clients, client1 and client2. Both of these two clients actually connected to the server but sent different messages. Figure 6.8 is the sequence diagram of the interaction among the server and the two clients. This sequence diagram only exemplifies a possible sequence of the events. The actual events sequence can vary depending on the run-time environment.

Three traces were captured when these three programs were running and interacting with each other. 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 consisting of $Server.trace$ and $Client1.trace$ is named $dual\_trace\_21$. The other consisting of $Server.trace$ and $Client2.trace$ is named $dual\_trace\_22$. I performed the "Stream Extraction" and "Communication Identification" operations for these two dual_traces with the functions descriptor in Table 6.4.

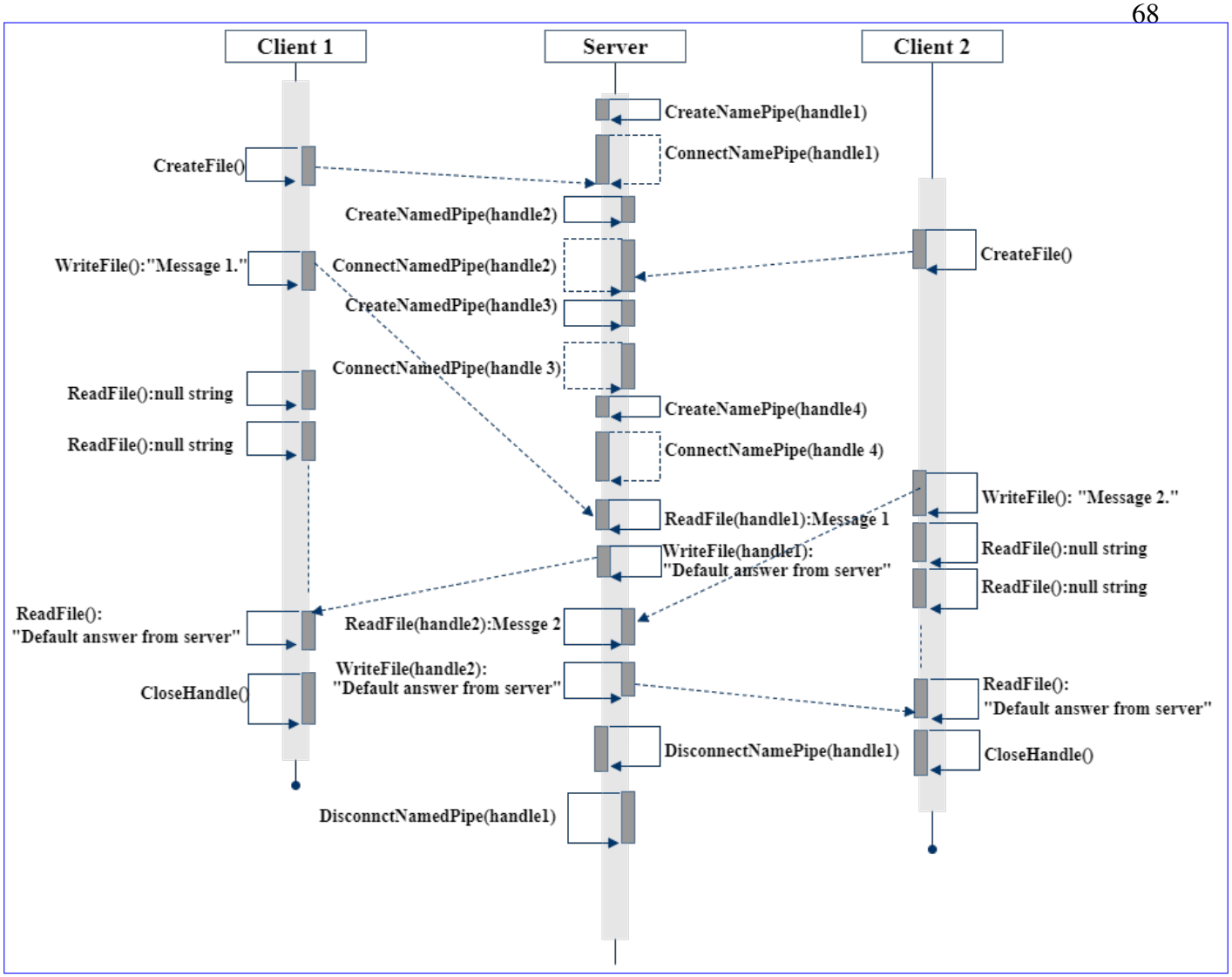

Figure 6.8: Sequence diagram of experiment 2

Table 6.4: Functions 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 Results Walk Through

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

6.2.2.1 $Server.trace$:

$Server.trace$ has 1,789,627 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 6.9.

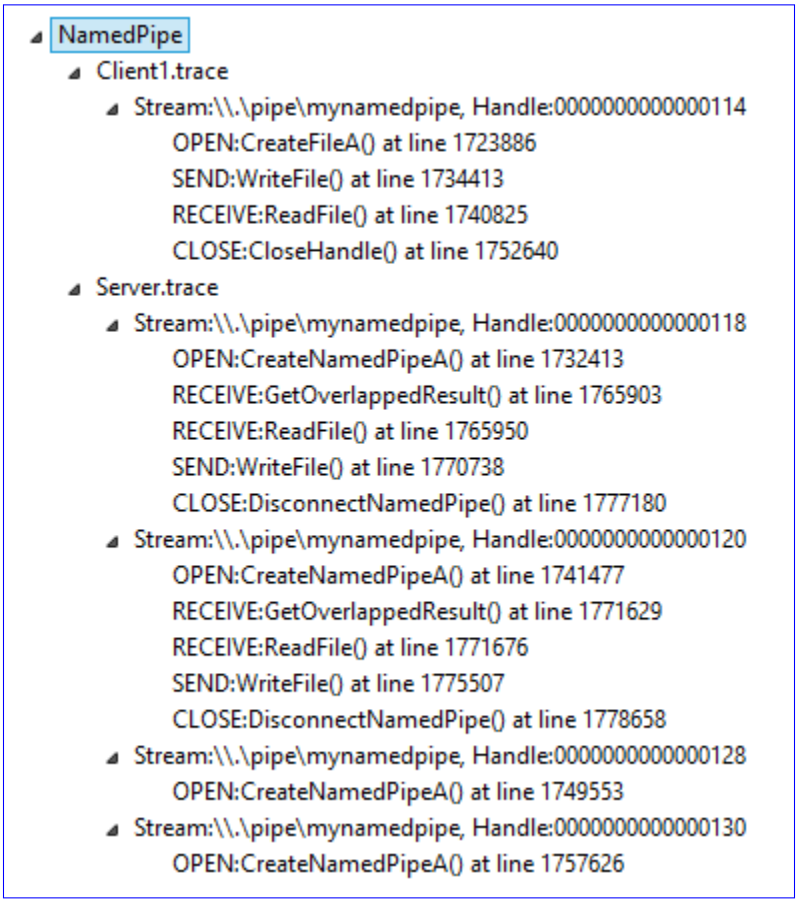

Figure 6.9: Extracted streams of $dual\_trace\_21$

6.2.2.2 $Client1.trace$:

$Client1.trace$ has 1,764,281 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 a stream 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 6.9.

6.2.2.3 $Client2.trace$:

$Client2.trace$ 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	$funN : WriteFile, type : send, inparams : \{Handle : 0x114, SendBuf : \text{“}Message\ 2\text{”}\}, outparams : \{Length : 15\}$
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 a stream was extracted, which consists of all these four function call events.

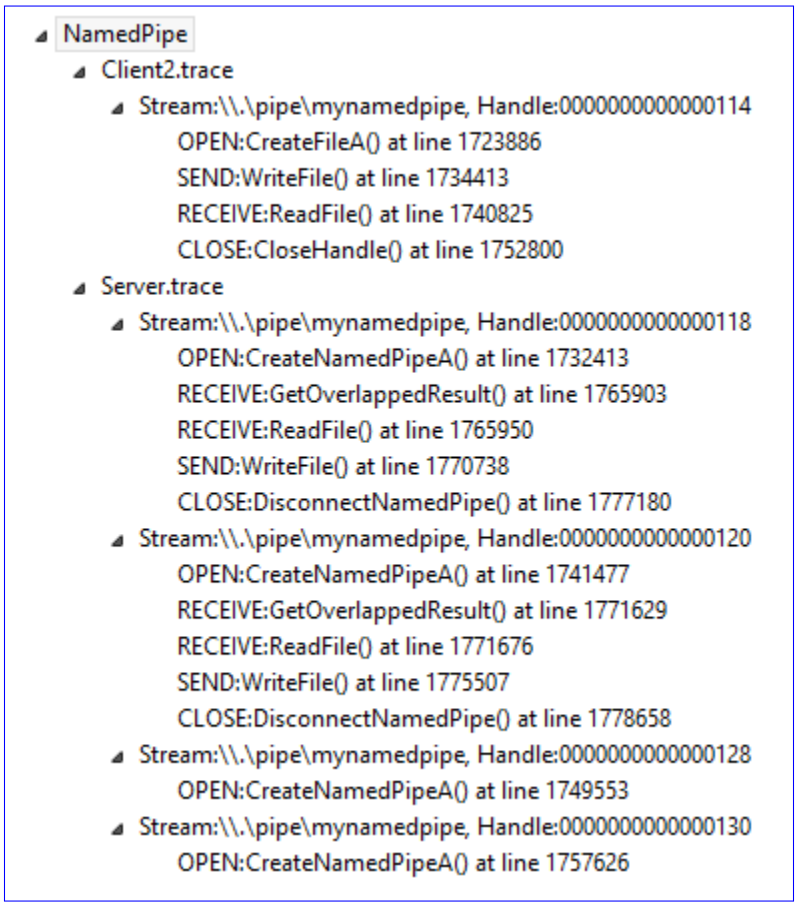

Figure 6.10: Extracted streams of $dual\_trace\_22$

6.2.2.4 $Dual\_trace\_21$:

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, the stream 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 shown in Table 6.8. Comparing the concatenation of each stream in $Server.trace$, it is obvious that the send payload concatenation of Stream $0x114$ in $Client1.trace$ matches the receive payload concatenation of Stream $0x120$ in $Server.trace$, while on the other direction, the send payload concatenation of Stream $0x120$ in $Server.trace$ matches the receive payload concatenation of Stream $0x114$ in $Client1.trace$, so, 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"	$WriteFile$: "$Default$ "	"$Default\ answer$ "
		$ReadFile$: "$Message$ 2"		$answer\ from\ server$"	$from\ server$"
	$0x120$	$GetOverlappedResult$: ""	"$Message$ 1"	$WriteFile$: "$Default$	"$Default\ answer$ "
		$ReadFile$: "$Message$ 1"		$answer\ from\ server$"	$from\ server$"
	$0x128$	-	-	-	-
	$0x130$	-	-	-	-
Client1.trace	$0x114$	$ReadFile$: "$Default$	"$Default\ answer$	$WriteFile$: "$Message$ 1"	"$Message$ 1"
		$answer\ from\ server$"	$from\ server$"		

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 6.11.

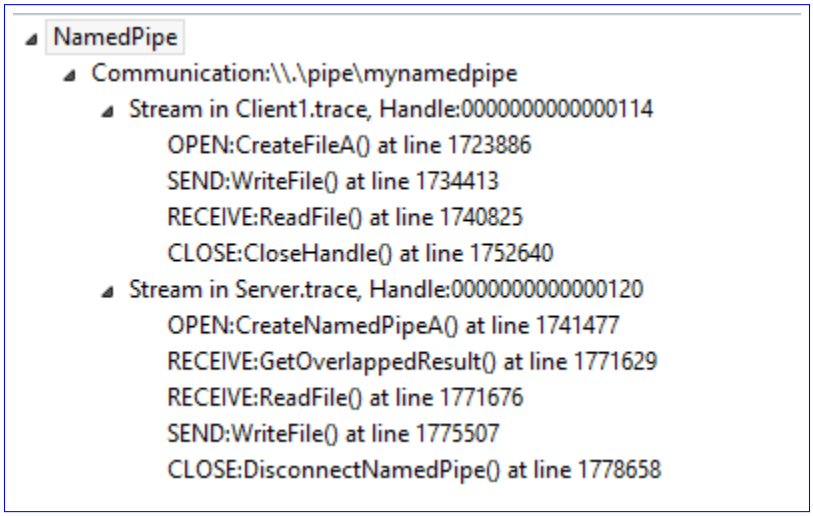

Figure 6.11: Identified communication of $dual\_trace\_21$

After I received the identified communication from $dual\_trace\_21$, I navigated from the send and receive events back to the traces. The navigation results are shown in Figure 6.12, Figure **??** and Figure **??**.

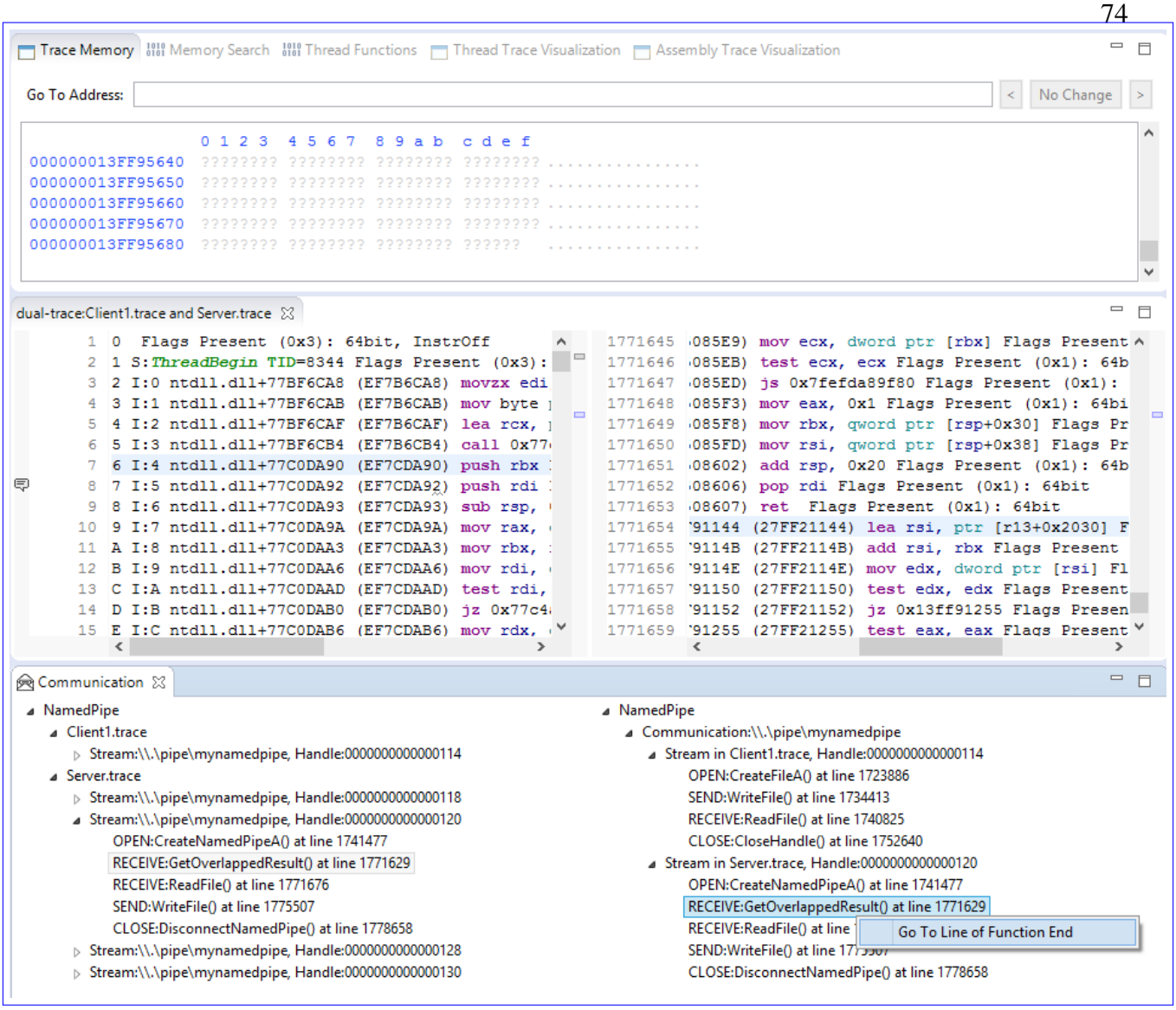

Figure 6.12: Navigation result for the function call event: $GetOverlappedResult$

Figure 6.12 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.

Figure 6.13 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 "$Message\ 1$".

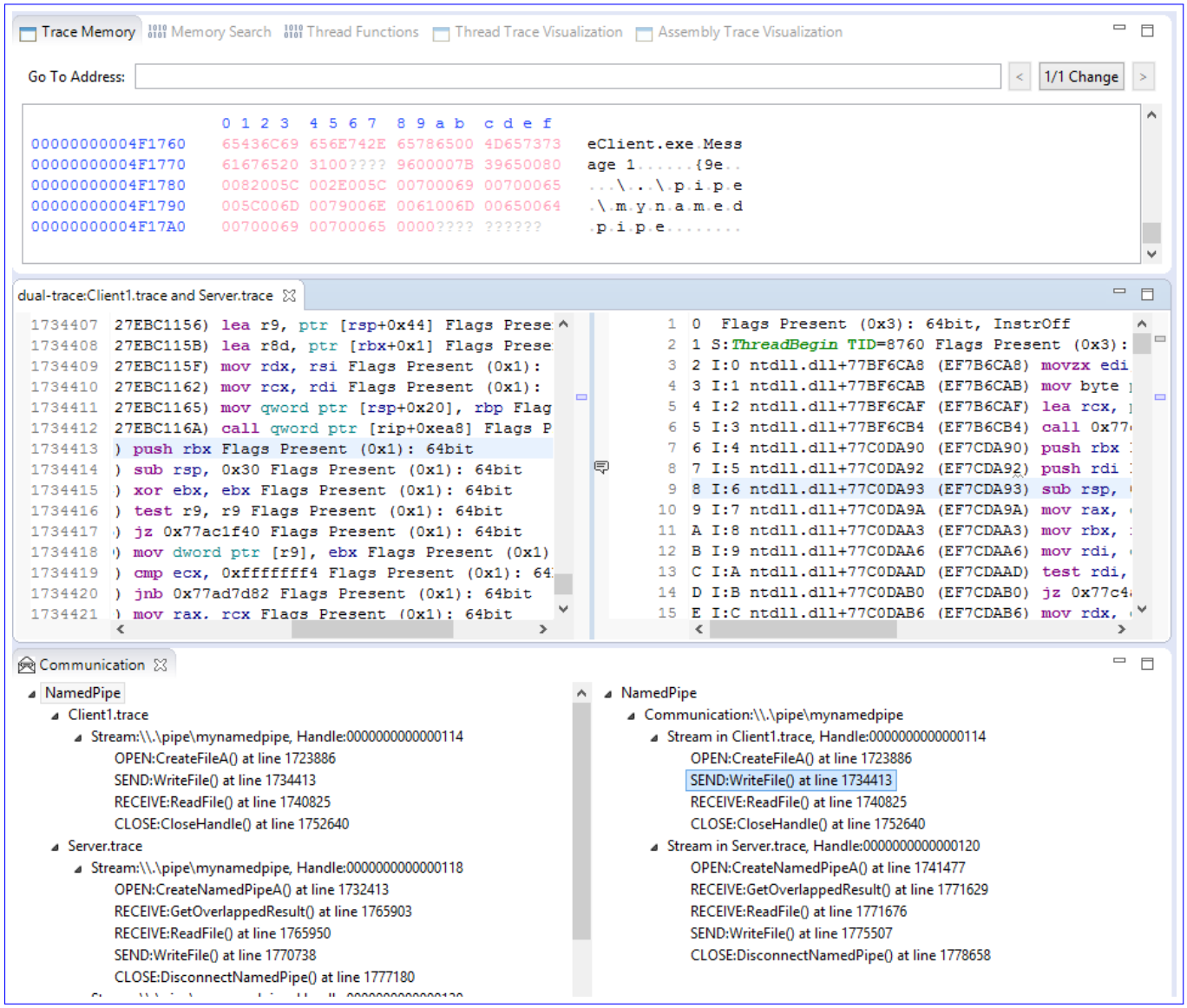

Figure 6.13: Client 1 send event navigation for the message "*Message 1*"

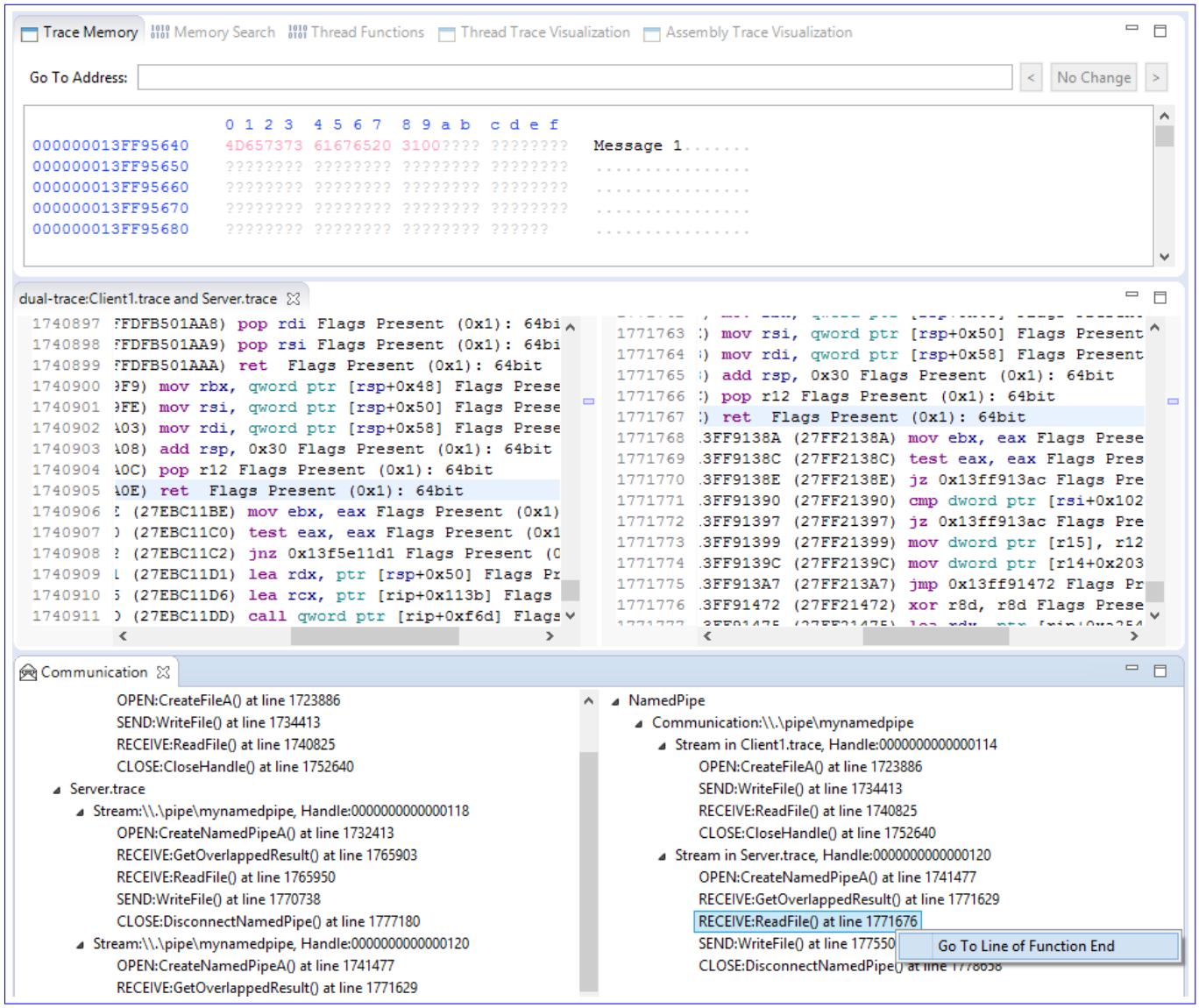

Figure 6.14: Sever receive event navigation for the message "*Message* 1"

~~Navigation results for the transmitted message "*Message* 1"~~

Figure 6.14 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 "$Message\ 1$" is transmitted from the client 1 to the server.

Figure 6.15 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 6.16 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 "$Default\ answer\ from\ server$" is transmitted from the server to the client 1.

6.2.2.5 $Dual\_trace\_22$:

Similar to $dual\_trace\_22$, all streams of $Server.trace$ will be matched to 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 stream in $Server.trace$, it is obvious that the send payload concatenation of Stream $0x114$ in $Client2.trace$ matches the receive payload concatenation of Stream $0x118$ in $Server.trace$, while in the other direction, the send payload concatenation of Stream $0x118$ in $Server.trace$ matches the receive payload concatenation of Stream $0x114$ in $Client2.trace$ so, only Stream $0x118$ of $Server.trace$ and Stream $0x114$ of $Client2.trace$ satisfy the content preservation of the reliable communication.

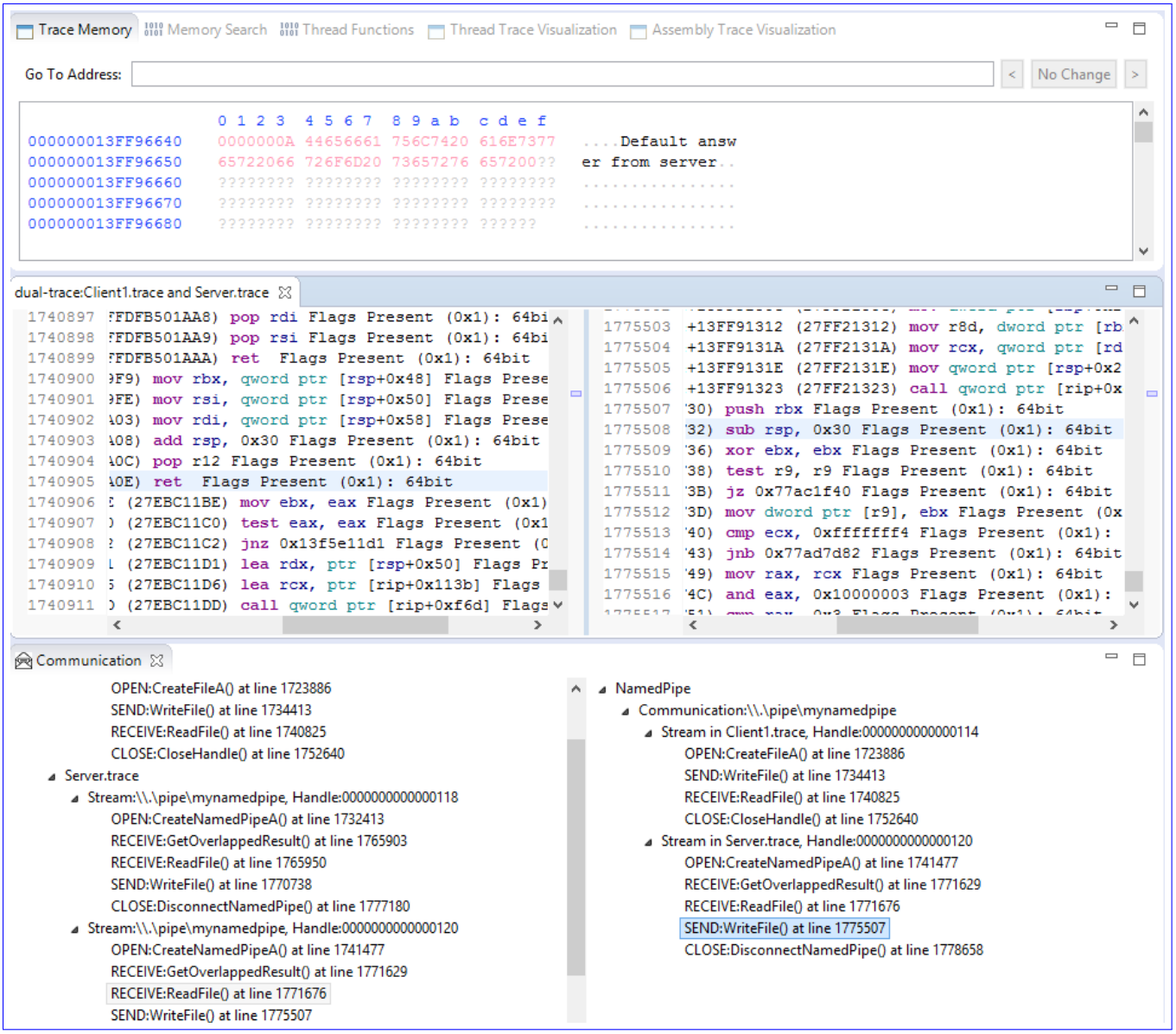

Figure 6.15: Server send event navigation for the message "*Default answer from server*"

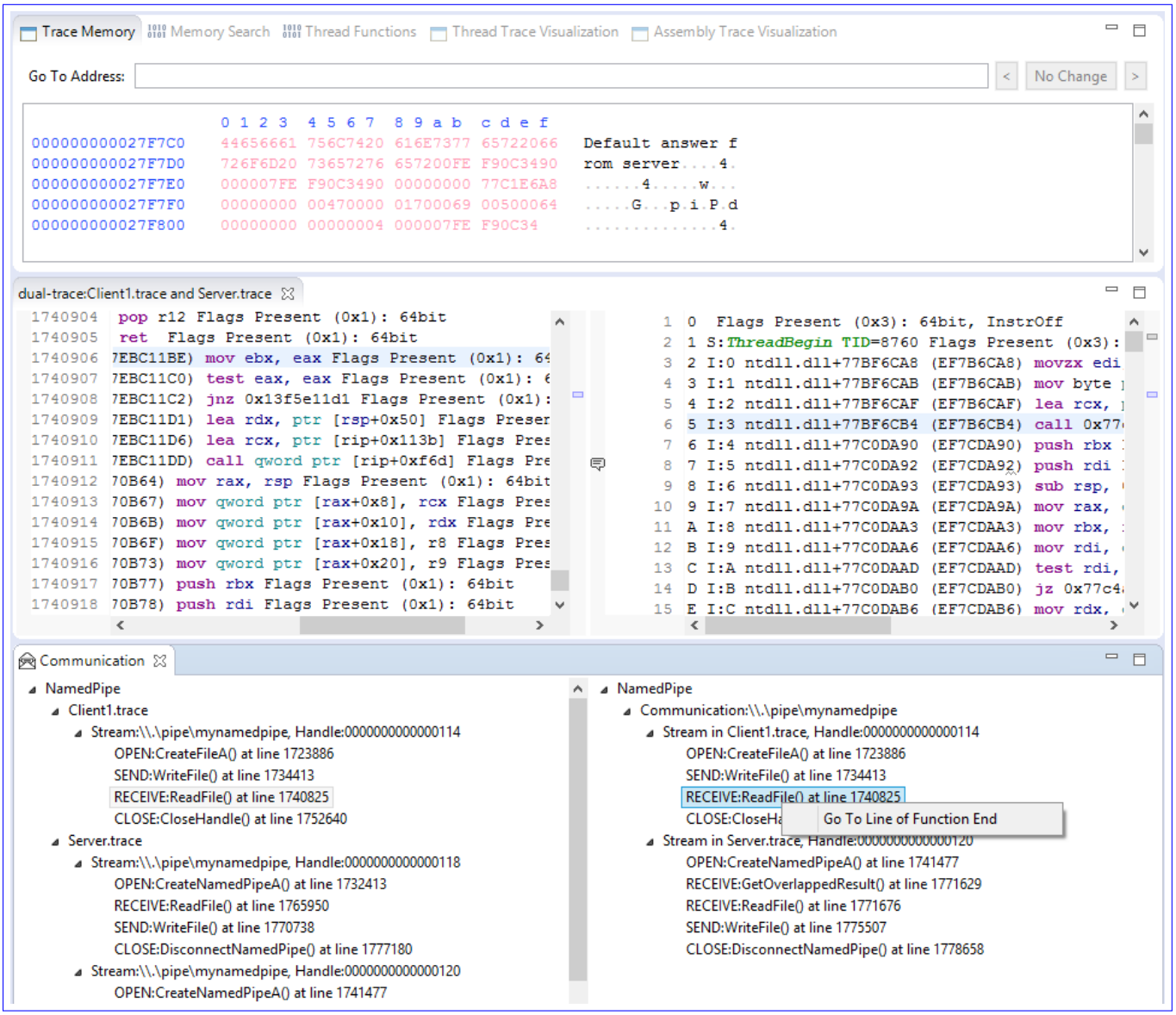

Figure 6.16: Client 1 receive event navigation for the message "*Default answer from server*"
~~Navigation results for the transmitted message "*Default answer from server*"~~

Table 6.9: Content summarize of the extracted streams

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

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 6.17.

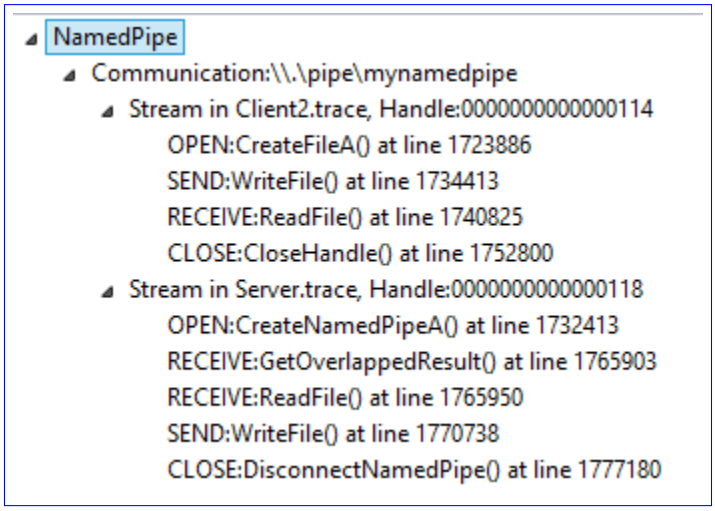

Figure 6.17: Identified communication of $dual\_trace\_22$

After I received the identified communication from $dual\_trace\_22$, I navigated from the send and receive events back to the traces. The navigation results are shown in Figure 6.18, Figure **??** and Figure **??**.

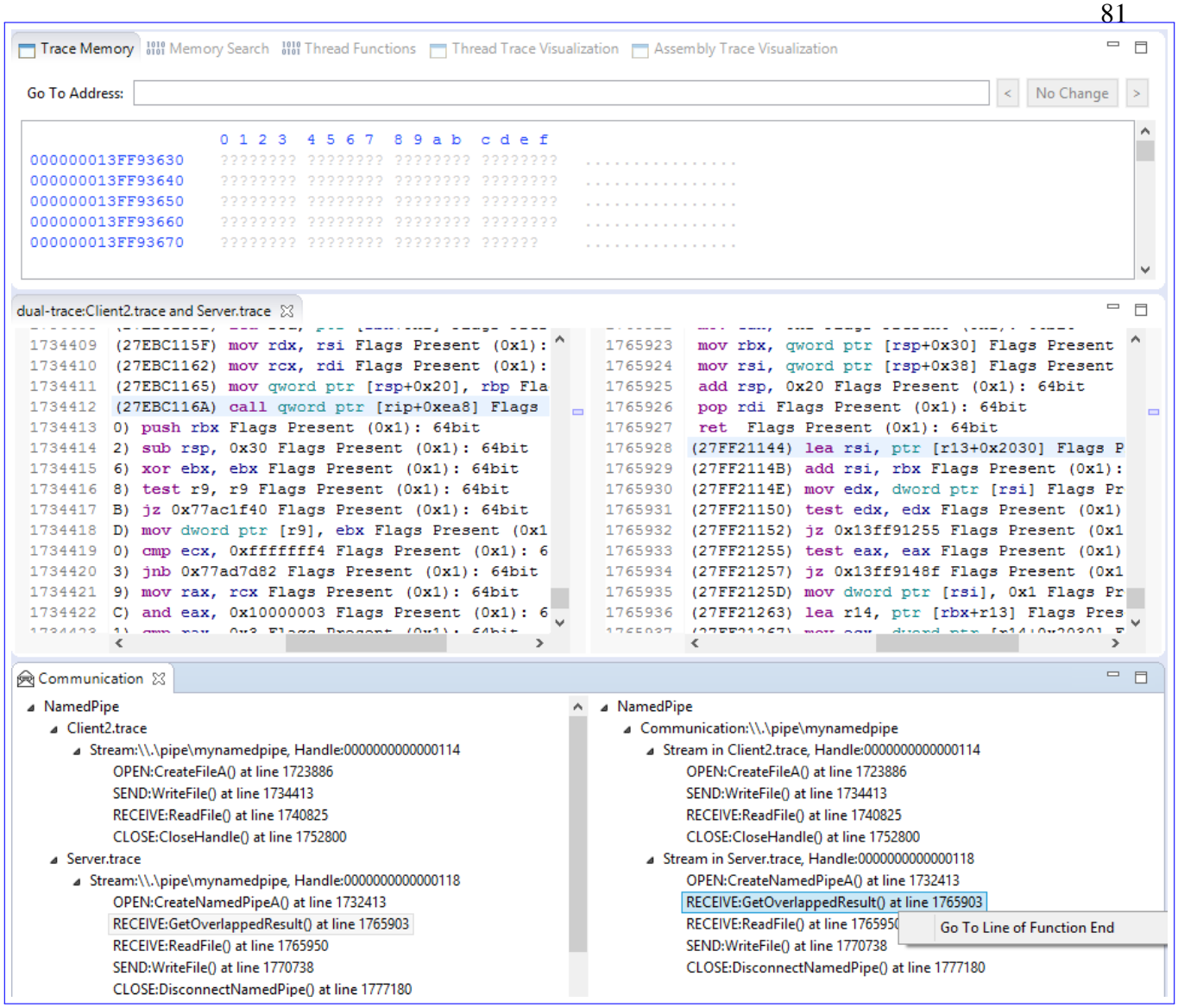

Figure 6.18: Navigation result for the function call event: $GetOverlappedResult$

Figure 6.18 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.

Figure 6.19 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 "$Message\ 2$".

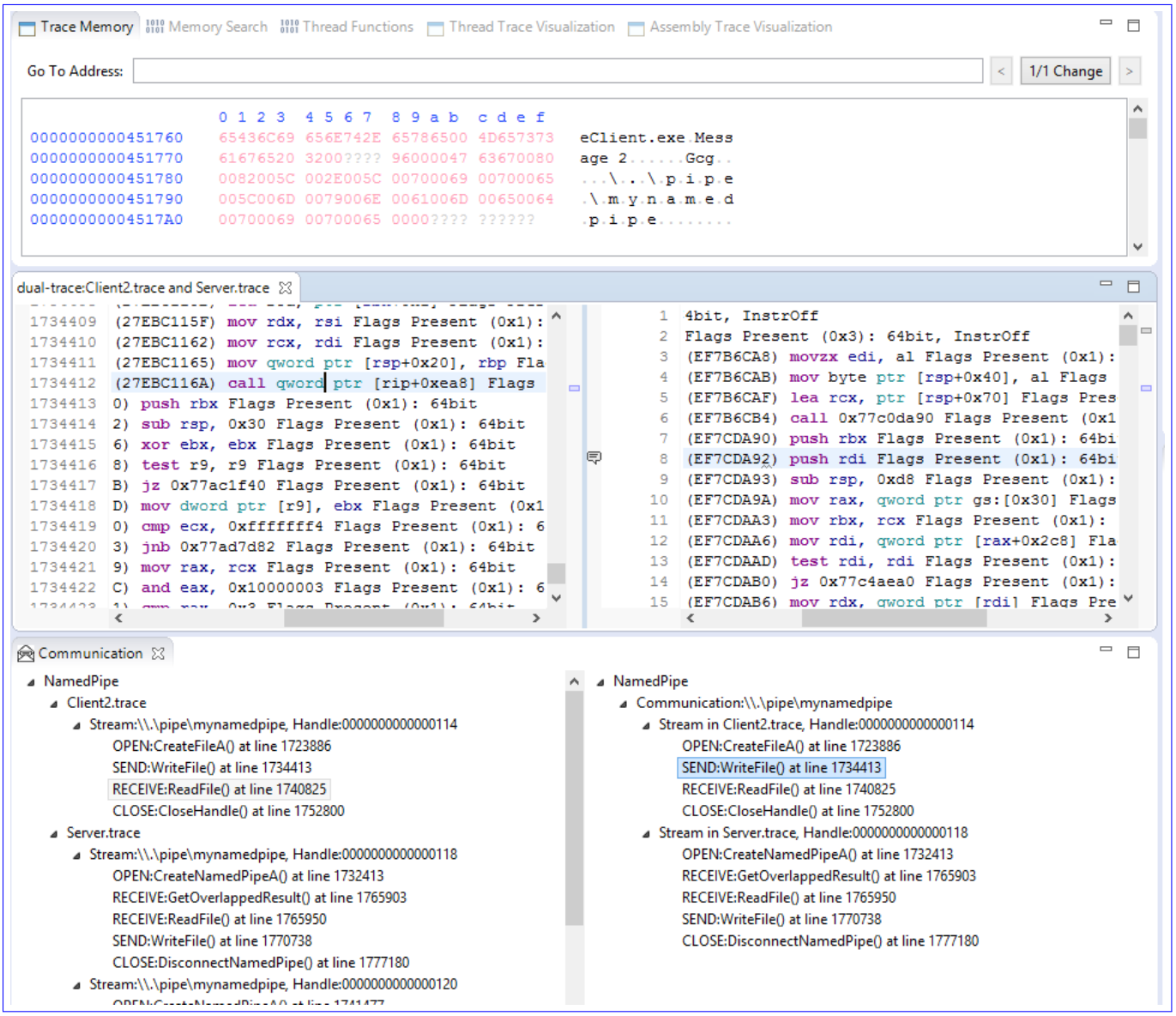

Figure 6.19: Client 2 send event navigation for the message "*Message 2*"

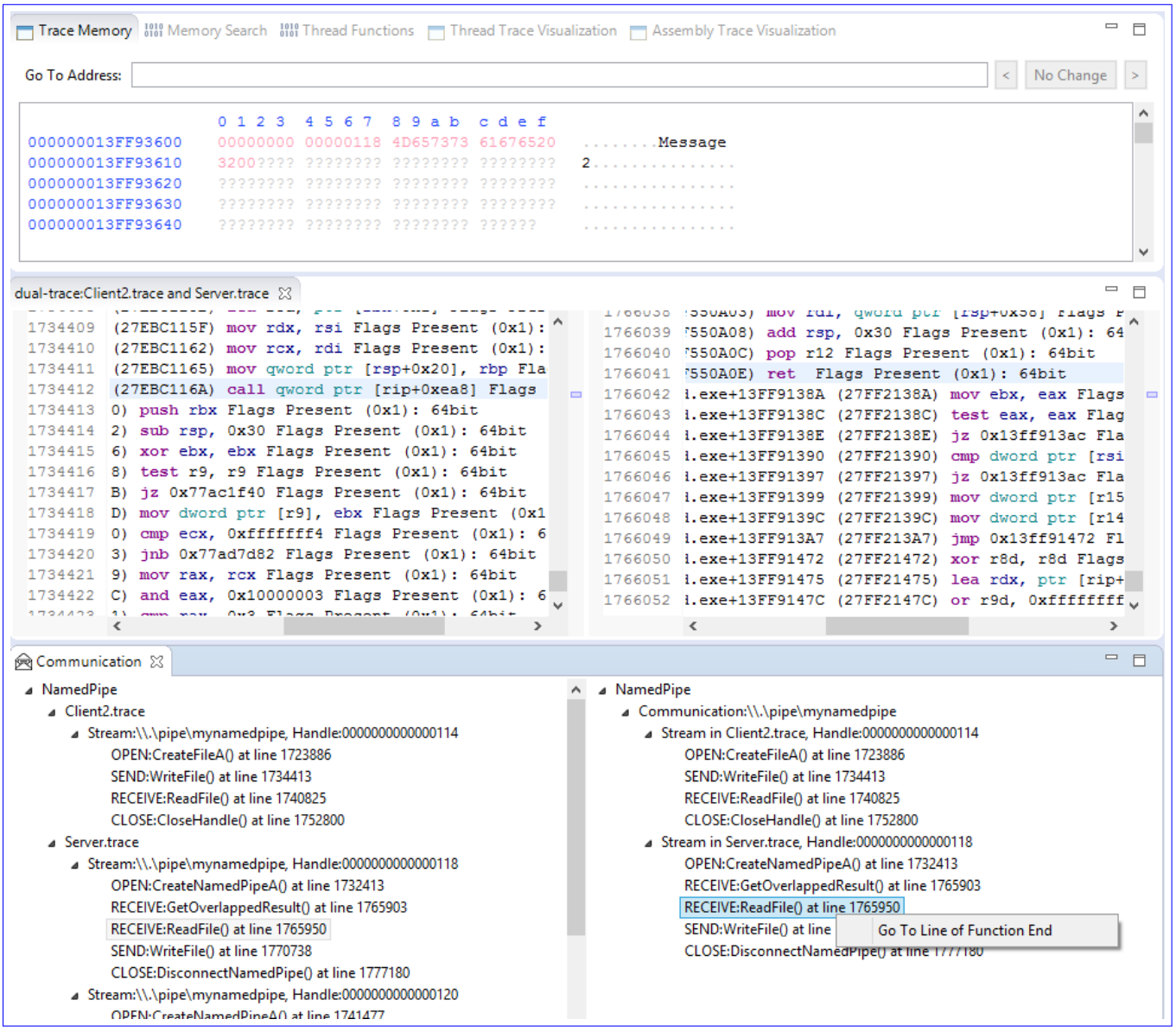

Figure 6.20: Sever receive event navigation for the message "*Message 2*"

~~Navigation results for the transmitted message "*Message 2*"~~

Figure 6.20 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 "$Message\ 2$" is transmitted from the client 2 to the server.

Figure 6.21 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 6.22 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 "$Default\ answer\ from\ server$".

These two figures perfectly show how the message "$Default\ answer\ from\ server$" is 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 transmitted can be shown in the "Trace Memory view" accurately.

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

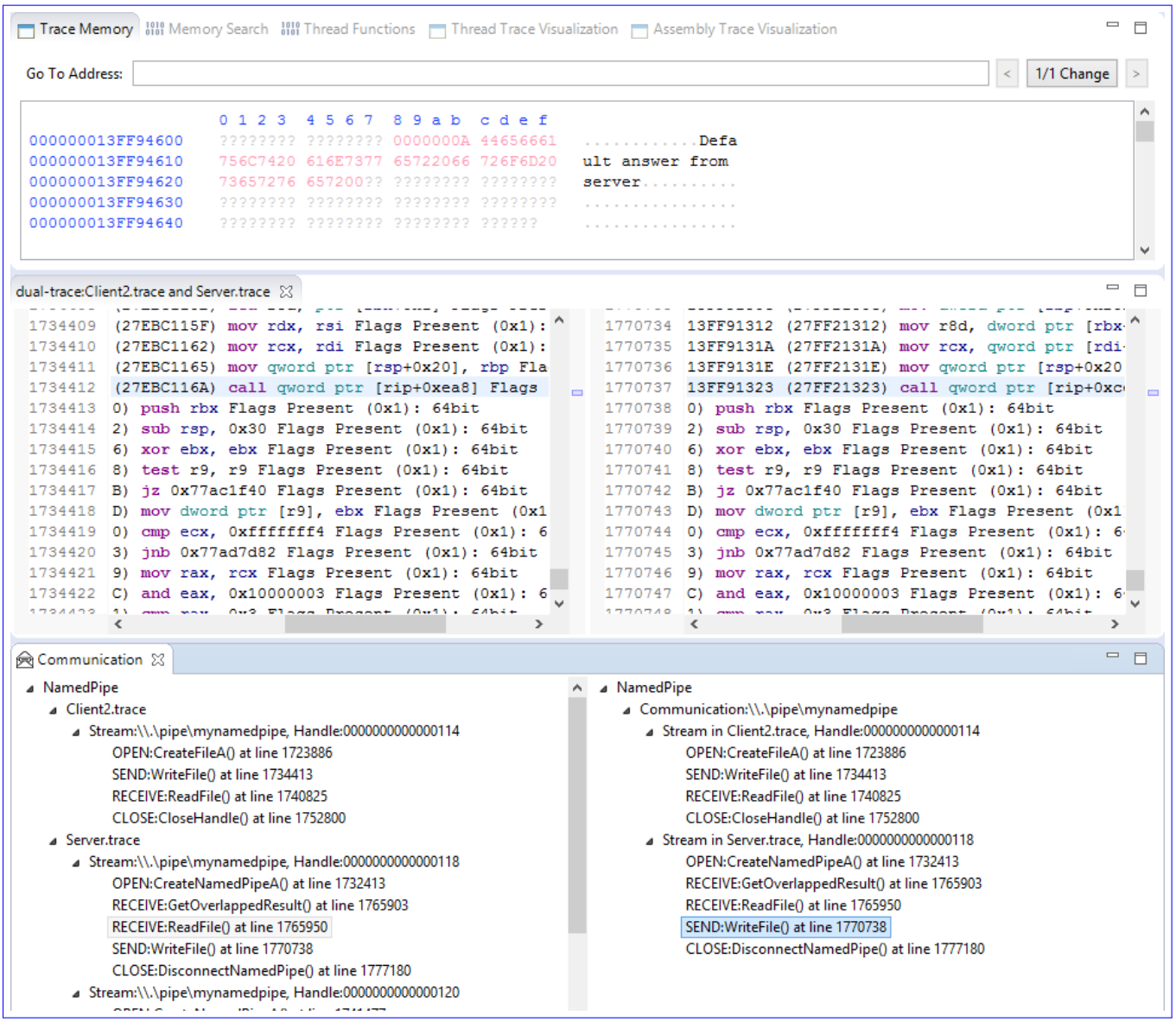

Figure 6.21: Server send event navigation for the message "*Default answer from server*"

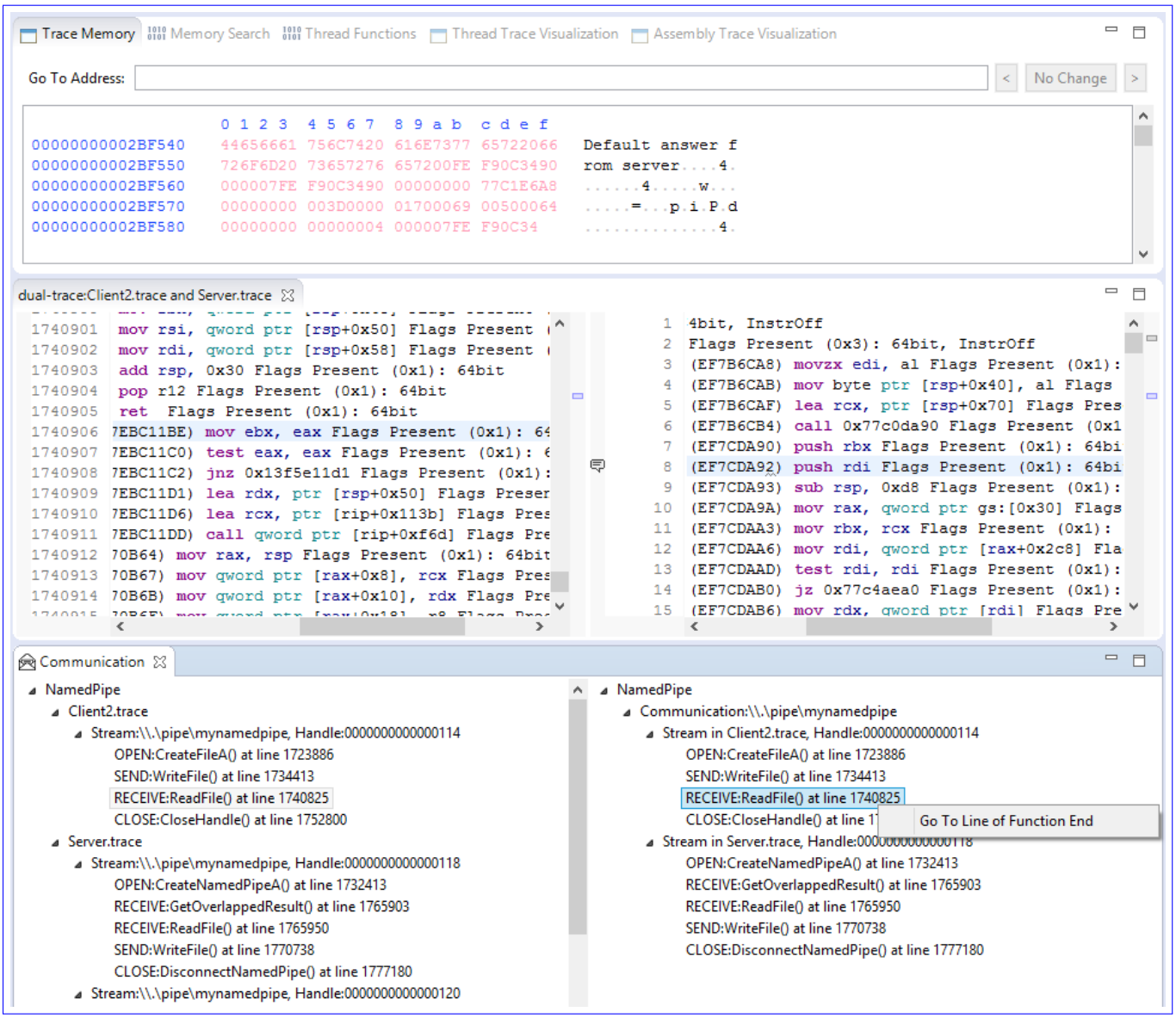

Figure 6.22: Client 2 receive event navigation for the message "*Default answer from server*"
~~Navigation results for the transmitted message "*Default answer from server*"~~

Chapter 7

Conclusions and Future Work

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

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 conduct 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 for developing other algorithms for other communication methods.

On top of the existing execution trace analysis environment, Atlantis, we implemented the communication identification features. These features provide the users a way to define the functions descriptor for communication methods through the configuration file. The user interface allows the users to conduct the communication identification and stream extraction from a dual trace and navigate back from the results to the views of the trace in Atlantis. A proof of concept demonstrate feasibility and usability.

7.1 Discussion

With the communication analysis prototype the user can reconstruct the communications easily. They need to understand the communication model to know what information they should retrieve

from the traces to reconstruct the communications. Then they have to understand the communication method they want to investigate and follow the functions descriptor formalization to develop the functions descriptor for this communication method. With the functions descriptor, the user actually can use the old Atlantis (without the implementation of this prototype) to perform the analysis. They need to search for all the function calls to the functions in the functions descriptor, then manually go through all the function calls in the search results and follow the communication identification approach to match the streams by checking the parameter values from the reconstructed memory state of the instruction lines. This manual analysis can be extremely tedious and exhausting. In some situation, there might be a lot of communication captured in the traces and each of these communications contain a lot of message exchanges. So manually performing the analysis could be an infeasible task.

The communication identified from the dual_trace give the necessary information, such as the transmitted data, the sequence of the packets of the communication. This provides the user a whole picture of a communication. This would be helpful for the users who need to understand how the data flow between the programs and might also help them to understand the architecture of the system.

There are two main limitation of this work: 1) the identified communications might be errors, and 2) the user need to specify the communication method for the analysis. This means the user must be an expert who understand all the possible communications that would happen between the two programs. For example, if the user doesn't indicate that the two interacting programs communicated through TCP sockets, the communications cannot be identified.

In conclusion, even though has limitations, this work is novel and shows its value for guiding the communication analysis through assembly-level execution traces. In addition, the prototype is the unique tool at the time of writing for the communication analysis in assembly-level.

7.2 Future Works

Future works includes:

- Extend the model to be more generalize for all kinds of interaction, 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 (user observation for tasks performing with the prototype and follow-up interview)

- Conduct an empirical study to properly evaluate the algorithms and implementation presented in this thesis (run some applications that contain known vulnerabilities that are related to communication and see if the communication identification can assist the analyst to detect these vulnerabilities.)

Accomplish of the first two future works can make the communication analysis method developed in this work more complete. The last two future works can validate the usefulness and Usability of this work.

Bibliography

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

[2] Derek Bruening. Qz: Dynamorio: Dynamic instrumentation tool platform. `http://www.dynamorio.org/`.

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

[4] Gerald Combs. Wireshark Go Deep. `https://www.wireshark.org/`.

[5] KAIST CysecLab. Codemap. `https://github.com/c0demap/codemap`.

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

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

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

[9] GCC. Options that control optimization. `https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Optimize-Options.html`.

[10] Michael Howard and David LeBlanc. *Writing secure code*. Pearson Education, 2003.

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

[12] Intel. Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[13]

[14] MSDN. Message queue (windows). `https://msdn.microsoft.com/en-us/library/ms705205(v=vs.85).aspx`.

[15] MSDN. Named pipes (windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365590(v=vs.85).aspx`.

[16] MSDN. Socket reference (windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/ms741416(v=vs.85).aspx`.

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

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

[19] Tcpdump. Tcpdump/Libpcap public repository. `http://www.tcpdump.org`.

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

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

[22] Oleh Yuschuk. Ollydbg. `http://www.ollydbg.de/`, 2007.

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

Appendix A

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"
```