Communication Detection and Data Transfer Event Synchronization from Dual Trace

by

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

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

MASTER OF SCIENCE

in the Department of Computer Science

by

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

Supervisory Committee

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

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

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

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

Supervisory Committee

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

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

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

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

ABSTRACT

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

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

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

Grant Organization Name, for funding me with a Scholarship.

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

Edith Wharton

DEDICATION

Just hoping this is useful!

Chapter 1

Modeling

In this chapter, I modeled the communication of two running programs. The dual-trace captured from two interacting programs are also modeled in the perspective of communication analysis. The modeling are based on the investigation of some common used communication methods. The communication methods are divided into two categories based on their data transmission properties. This modeling are the foundation to decide how communications being identified from the dual-trace and how to present them to the user. The terminology of using in this chapter can be found in 4.1.

1.1 Communication Categorization and Communication Methods

The goal of this work is to identify the communications from the dual-trace. We need to understand the properties of the communications to identify them. In general, there are two types of communication: reliable and unreliable in the terms of their reliability of data transmission. The reason to divide the communication methods into these two categories is that the data transmission properties of the communications fall in different categories affect the mechanism of the data verification in the identification algorithm. In the following two subsections, I summarize the characteristics of these two communication categories. The communication methods list in Table1.2 will be discussed further to provide more concrete comprehension.

Table 1.1: Communication Methods Discussed in This Work

Reliable Communication	**Unreliable Communication**
Named Pipes	Message Queue
TCP	UDP

1.1.1 Reliable Communication

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

1.1.2 Unreliable Communication

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

1.1.3 Communication Methods

In this section, I describe the mechanism and the basic data transfer characteristics of each communication method in Table1.2 briefly. Moreover, data transfer scenarios are represented correspondingly in diagrams for each communication method.

Named Pipe

A named pipe provides FIFO communication mechanism for inter-process communication. It can be one-way or duplex pipe which allows two programs send and receive message through the named pipe. [2]

The basic data transfer characteristics of Named Pipe are:

- Bytes received in order
- Bytes sent as a whole trunk can be received in segments
- No data duplication
- Only the last trunk can be lost

Based on these characteristics, the data transfer scenarios of Named pipe can be summarized in Figure1.1.

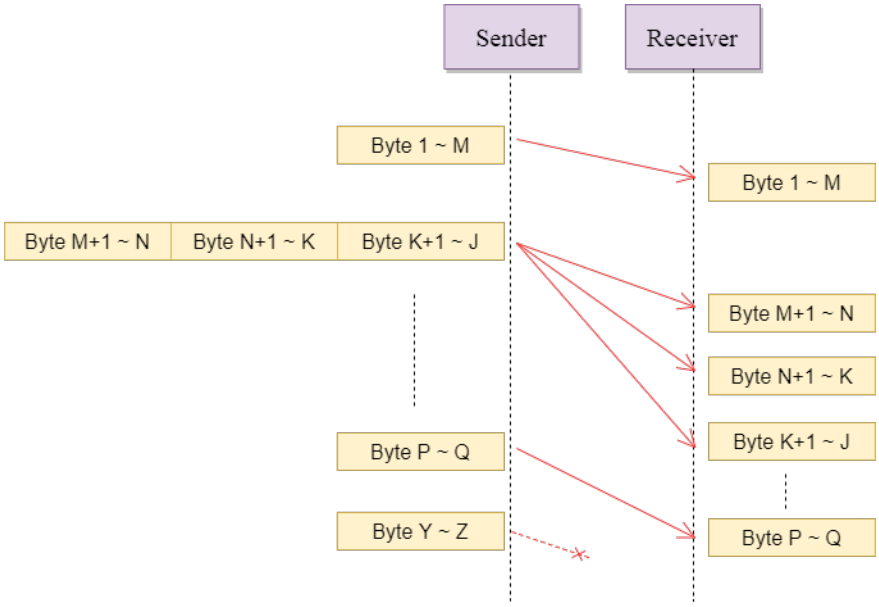

Figure 1.1: Data Transfer Scenarios for Named Pipe

Message Queue

Message Queuing (MSMQ) is a communication method to allow applications which are running at different times across heterogeneous networks and systems that may be temporarily offline can still communicate with each other. Messages are sent to and read from queues by applications. Multiple sending applications can send messages to and multiple receiving applications can read messages from one queue.[4] In this work, only one sending application versus one receiving application case is considered. Multiple senders to multiple receivers scenario can be divided into multiple sender and receiver situation. Both applications of a communication can send to and receive from the channel.

The basic data transfer characteristics of Message Queue are:

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

Based on these characteristics, the data transfer scenarios of Message Queue can be summarized in Figure1.2.

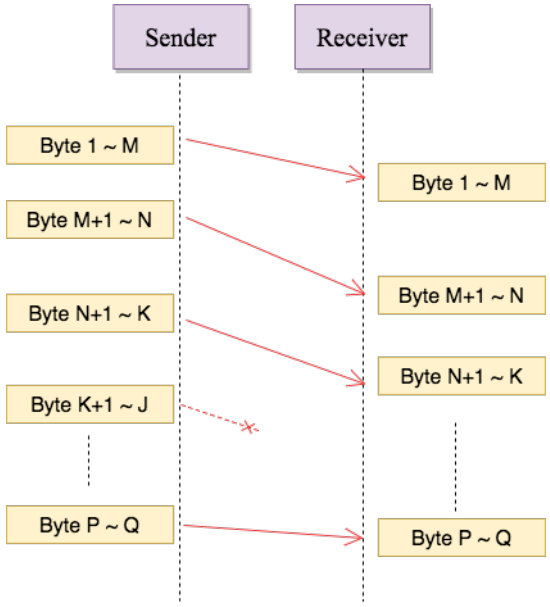

Figure 1.2: Data Transfer Scenarios for Message Queue

TCP

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

The basic data transfer characteristics of TCP are:

- Bytes received in order
- No data lost(lost data will be re-transmitted)

- No data duplication
- Sender window size is different from receiver's window size, so packets can be re-segmented

Based on these characteristics, the data transfer scenarios of TCP can be summarized in Figure1.3.

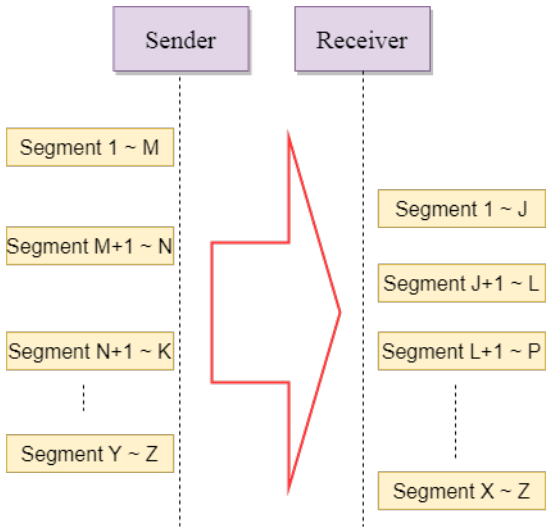

Figure 1.3: Data Transfer Scenarios for TCP

UDP

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

The basic data transfer characteristics of UDP are:

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

Based on these characteristics, the data transfer scenarios of UDP can be summarized in Figure1.4.

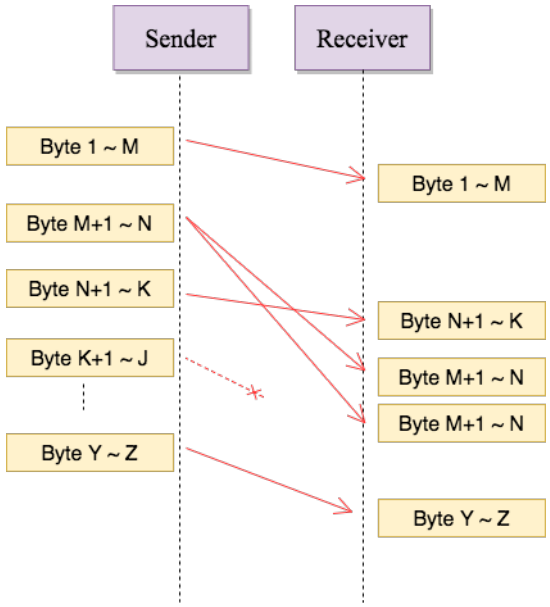

Figure 1.4: Data Transfer Scenarios for UDP

1.2 Communication Model

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

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

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

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

Preservation for reliable communication:

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

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

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

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

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

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

Figure1.5 is an example of the reliable communication. In this example, $ss_{0,0}$ = "ab", $ss_{0,1}$ = "cde", $ss_{0,2}$ = "fgh"; $sr_{1,0}$ = "abc", $sr_{1,1}$ = "def", $ss_{1,2}$ = "gh" and on the other direction $ss_{1,0}$ = "mno", $ss_{1,1}$ = "pqr", $ss_{1,2}$ = "stu"; $sr_{0,0}$ = "$mnop$", $sr_{0,1}$ = "$qrstu$". It is clear in the example that$ss_{0,0}.ss_{0,1}.ss_{0,2} = sr_{1,0}.sr_{1,1}.ss_{1,2}$ = "$abcdefgh$" and $ss_{1,0}.ss_{1,1}.ss_{1,2} = sr_{0,0}.sr_{0,1}$ = "$mnopqrstu$". These satisfy the content preservation. The timing in this example are: $ts_{1,0} < ts_{1,1} < tr_{0,0} < ts_{1,2} < tr_{0,1}$ and $ts_{0,0} < ts_{0,1} < tr_{1,0} < ts_{1,2} < tr_{1,1} < tr_{1,2}$. The following fact of this example satisfy the timing preservation. $sr_{0,0}$ = "$mnop$" is the sub string of $ss_{1,0}.ss_{1,1}$ = "$mnopqr$", $sr_{0,0}.sr_{0,1}$ = "$mnopqrstu$" is the sub string of $ss_{1,0}.ss_{1,1}.ss_{1,2}$ = "$mnopqr$"stu, $sr_{1,0}$ = "abc" is the sub string of $ss_{0,0}.ss_{0,1}$ = "$abcde$", $sr_{1,0}.sr_{1,1}$ = "$abcdef$" and $sr_{1,0}.sr_{1,1}.sr_{1,2}$ = "$abcdefgh$" are the sub string of $ss_{0,0}.ss_{0,1}.ss_{0,2}$ = "$abcdefg$".

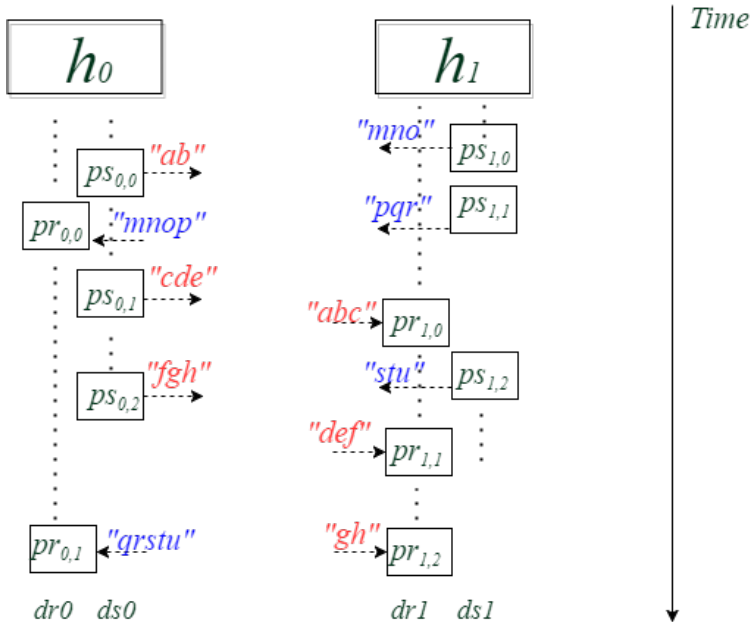

Figure 1.5: Example of Communication

Figure1.6 is an example of the unreliable communication. In this example, $sr_{1,0} = ss_{0,1} =$ "cde", $tr_{1,0} > ts_{0,1}$; $sr_{1,1} = ss_{0,2} =$ "fi",$tr_{1,1} > ts_{0,2}$; $sr_{0,0} = ss_{1,0} =$ "gh", $tr_{0,0} > ts_{1,0}$; $sr_{0,1} = ss_{1,1} =$ "$ijklm$", $tr_{0,1} > ts_{1,1}$; $sr_{0,2} = ss_{1,2} =$ "n", $tr_{0,2} > ts_{1,2}$. All of these satisfy the content preservation and timing preservation of the unreliable communication.

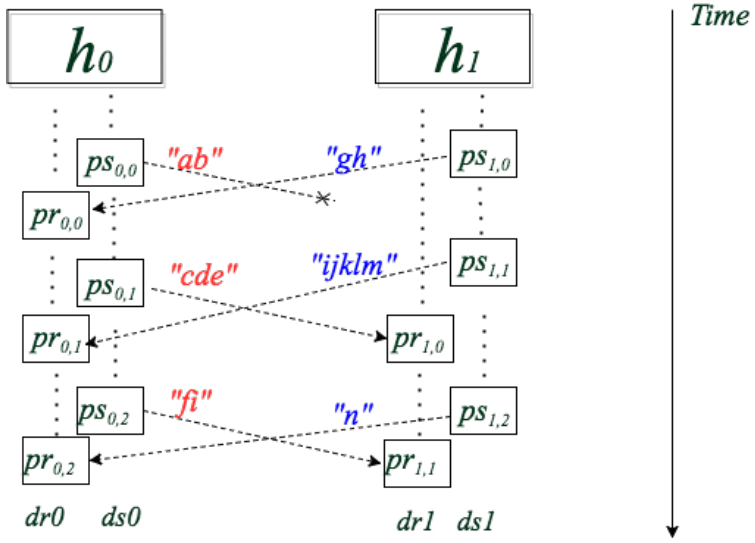

Figure 1.6: Example of Communication

1.3 Dual-Trace Model

In this section, I model the execution trace in the dual-trace. The modeling aims at identify the communications from the information summarized in the model.

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

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

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

1.4 Element Matching Communication Model and Dual-Trace Model

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

Table 1.2: Element Matching of Communication and Trace Models

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

Chapter 2

Communication Identification Algorithms

This chapter discuss the algorithms for communication identification from dual-trace. Pseudo code are listed for algorithms. The algorithm is based on the models developed in the models Chapter1.

2.1 Communication Identification Algorithm

The identification of the communications from a dual_trace should be able to identify the concerned communications as well as all the components defined in it. The inputs of this algorithm are the $dual\_trace = \{trace_x : x = 0, 1\}$ and the concerned communication method's function set $funcset = \{func_l, 0 \leqslant l \leqslant L\}$. The output of this algorithm is all the identified communications of the concerned communication method. This is a very high level algorithm, details of each step in this algorithm will be discussed in the later sections.

Algorithm 1: Communication Identification Algorithm

Input: $dual\_trace$, $funcset$

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

for $x \in (0, 1)$ **do**
    $event\_trace_x = eventfilter\,(trace_x, funcset)$;
    $stream\_trace_x = streamfilter\,(event\_trace_x)$;
$cos = streammatch(stream\_trace_0, stream\_trace_1)$;
return cos;

2.2 Communication Methods' Implementation in Windows

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

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

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

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

2.2.1 Windows Calling Convention

The Windows calling convention is important to know in this research. The communication identification relies not only on the system function names but also the key parameter values. In the assembly level execution traces, the parameter values is captured in the memory changes of the instructions. The memory changes are recognized by the register names or the memory address. The calling convention helps us to understand where the parameters are stored so that we can find them in the memory change map in the trace. Calling Convention is different for operating systems and the programming language. The Microsoft* x64 example calling convention is listed in 4.2 since we used dual-trace from Microsoft* x64 for case study in this work.

2.2.2 Named Pipes

In Windows, a named pipe is a communication method for the pipe server and one or more pipe clients. The pipe has a name, can be one-way or duplex. Both the server and clients can read or write into the pipe.[3] In this work, I only consider one server versus one client communication. One server to multiple clients scenario can always be divided into multiple server and client communications thanks to the characteristic that each client and server communication has a separate conduit. The server and client are endpoints in the communication. We call the server "server endpoint" while the client "client endpoint". The server endpoint and client endpoint of a named pipe share the same pipe name, but each endpoint has its own buffers and handles.

There are two modes for data transfer in the named pipe communication method, synchronous and asynchronous. Modes affect the functions used to complete the send and receive operation. I list the related functions for both synchronous mode and asynchronous mode. The create channel functions for both modes are the same but with different input parameter value. The functions for send and receive message are also the same for both cases. However, the operation of the send and receive functions are different for different modes. In addition, an extra function *GetOverlappedResult* is being called to check if the sending or receiving operation finish, the output message will be stored in the overlap structure whose memory address saved in the function's output parameter Overlap Structure Address. Table2.1 lists the functions of the events for synchronous mode while Table2.2 lists the functions of the events for the asynchronous mode for a Named pipe communication.

Table 2.1: Function List of events for Synchronous Named Pipe

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

Table 2.2: Function List of events for Asynchronous Named Pipe

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

A named pipe server is responsible for the creation of the pipe, while clients can connect to the pipe after it was created. The creation and connection of a named pipe returns the handle ID of that pipe. These handler Ids will be used later when data is being sent or received to a specified pipe. Figure2.1 shows the channel set up process for a Named Pipe communication.

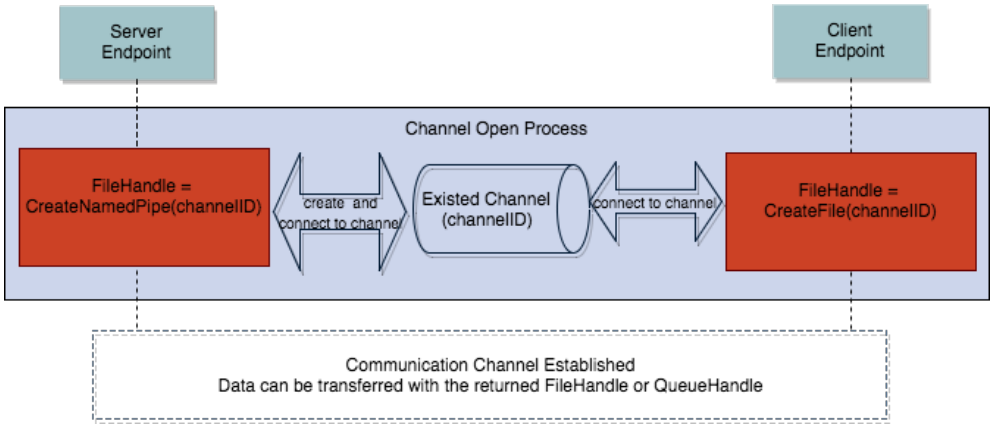

Figure 2.1: Channel Open Process for a Named Pipe

2.2.3 Message Queue

Similar to Named Pipe, Message Queue's implementation in Windows also has two modes, synchronous and asynchronous. Moreover, the asynchronous mode further divides into two operations, one with callback function while the other without. With the callback function, the callback function would be called when the send or receive operations finish. Without callback function, the general function *MQGetOverlappedResult* should be called by the endpoints to check if the message sending or receiving operation finish, the output message will be stored in the overlap structure whose memory address saved in the function's output parameter Overlap Structure Address. Table2.3 lists the functions for synchronous mode while Table2.4 and Table2.5 list the functions for the asynchronous mode with and without callback.

Table 2.3: Function List of events for Synchronous MSMQ

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

Table 2.4: Function List of events for Asynchronous MSMQ with Callback

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

Table 2.5: Function List of events for Asynchronous MSMQ without Callback

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

The endpoints of the communication can create the queue or use the existing one. However, both of them have to open the queue before they access it. The handle ID returned by the open queue function will be used later on when messages are being sent or received to identify the queue. Figure2.2 shows the channel set up process for a Message Queue communication.

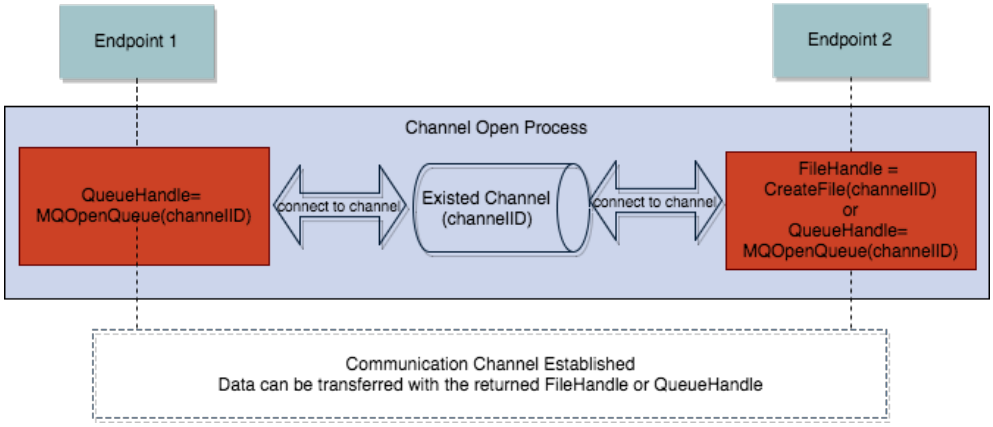

Figure 2.2: Channel Open Process for a Message Queue

2.2.4 TCP and UDP

In Windows programming, these two methods shared the same set of APIs regardless the input parameter values and operation behaviour are different. In Windows socket solution, one of the two endpoints is the server while the other one is the client. Table 2.6 lists the functions of a UDP or TCP communication.

Table 2.6: Function List of events for TCP and UDP

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

The communication channel is set up by both of the endpoints. The function *socket* should be called to create their own socket on both endpoints. After the sockets are created, the server endpoint binds the socket to its service address and port by calling the function *bind*. Then the server endpoint calls the function *accept* to accept the client connection. The client will call the function *connect* to connect to the server. When the function *accept* return successfully, a new socket handle will be generated and returned for further data transfer between the server endpoint and the connected client endpoint. After all these operations are performed successfully, the channel is established and the data transfer can start. During the channel open stage, server endpoint has two socket handles, the first one is used to listen to the connection from the client, while the second one is created for real data transfer. Figure2.3 shows the channel open process for TCP and UDP.

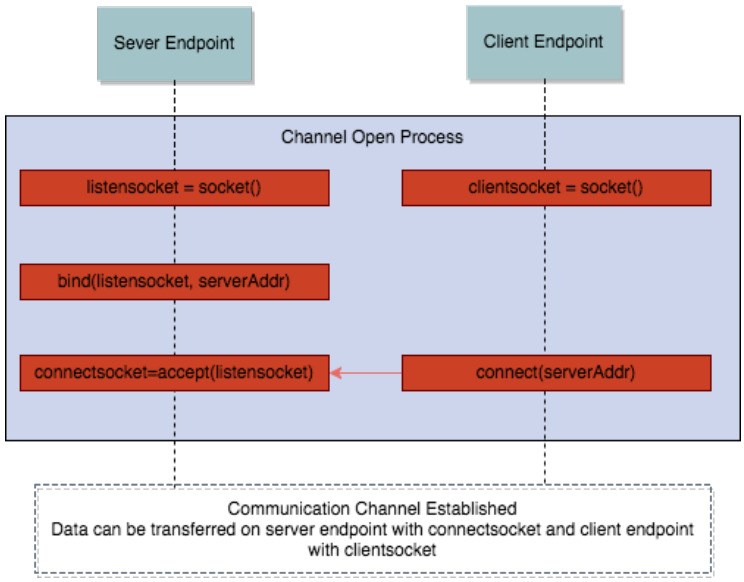

Figure 2.3: Channel Open Model for TCP and UDP

2.3 Event Locating Algorithm: $eventfilter\ ()$

The concerned events in a communication are channel open, channel close, send and receive events. These events are identified as system function calls in this work. A function call in the trace starts from the function call instruction to the function return instruction. The input parameters' value and input buffer content should be retrieved from the memory state of the the function call instruction line while the return value, output parameters' value and output buffer content should be retrieved from the memory state of the function return instruction line. Tables in section 2.2 indicate all the functions of the communication methods as well as the concerned parameters. Following the windows calling convention, the concerned parameter value or buffer address can be found in the corresponding register or stack positions. The buffer content can be found in the memory address in the reconstructed memory state. Each event can be completed by different function calls. For example, for the client endpoint in TCP communication method, both *socket* and *connect* function call are considered to be the channel open events. The functions list for a communication method is needed as a input of this algorithm. Tables in Section2.2 give the examples of function list of the events for some communication methods. The algorithm presented in this section is designed for locating all function calls provided in the function list as events of one communications method. If more than one communication methods are being investigated, this algorithm should be run multiple times, each for a method. Events in the output event list is sorted by time of occurrence.

Since the function list usually contain a very small number of functions compared to the instruction line number in the execution trace, the time complexity of this algorithm is O(N+M) , N and M are the instruction line numbers of the two traces in the duel-trace.

Algorithm 2: Event Locating Algorithm

Input: $trace, funcset$

Output: $event\_trace$

```
1  event_trace ← List⟨Event⟩;
2  while not at end of trace do
3      for f ∈ funcset do
4          if Is function call of f then
5              event.funN = f.funN event.startline ← current Line number;
6              event.endline ← find function return instruction line;
7              event.inputs ← reconstruct memory of event.startline from the trace and
                get input values of f.pars;
8              event.outputs ← reconstruct memory of event.endline from the trace and get
                outputs values of f.pars;
9              event.type ← f.type;
10             event_trace.add (event);
11 return event_trace;
```

2.4 Stream Identification Algorithm: $streamfilter\ ()$

The events located in the $event\_trace$ may correspond to different $stream$, the next step in the communication identification algorithm is to identify them for each $stream$. The input of this algorithm the $event\_trace$ from the "Event Locating Algorithm". Since the input $event\_trace$ is sorted by time of occurrence and the channel open events should always happen before other events, it is reasonable to assume the new stream can be identified by its first channel open function call. The identification for TCP and UDP server endpoints are slightly complicated than the other ones, due to its own channel open mechanism. The output of this algorithm is the $stream\_trace$. Each stream in this $stream\_trace$ consist of the sub streams. The concepts of the stream and sub streams are defined in Section4.1.

Algorithm 3: Stream Indentification Algorithm

Input: $event\_trace$

Output: $stream\_trace$

$stream\_trace \leftarrow Map\langle String, List\langle EndPoint\rangle\rangle$;
for $event \in event\_trace$ **do**
    if $event$ *is a channel open event* **then**
        $handle \leftarrow$ get the handle identifier from the function parameter list;
        $stream \leftarrow stream\_trace.get\,(handle)$;
        if $event$ *is an accept* $(event)$ *function call for TCP or UDP* **then**
            $newHandle \leftarrow$ get the second socket handle identifier which is the return value from the function parameter list;
            $stream\_trace.remove\,(handle)$;
            $stream\_trace.add\,(newHandle, endpoint)$;
        if $endpoint$ *is null* **then**
            $stream = New\ \ Stream\,()$;
            $stream\_trace.add\,(hanele, endpoint)$;
        $stream.openStream.add\,(event)$;
    if $event$ *is a channel send event* **then**
        $handle \leftarrow$ get the handle from the function parameter list;
        $stream \leftarrow stream\_trace.get\,(handle)$;
        if $stream$ *is not* $null$ *and* $stream.complete$ *is* $False$ **then**
            $stream.sendStream.add\,(event)$;
    if $event$ *is a channel receive event* **then**
        $handle \leftarrow$ get the handle from the function parameter list;
        $stream \leftarrow stream\_trace.get\,(handle)$;
        if $stream$ *is not* $null$ *and* $stream.complete$ *is* $False$ **then**
            $stream.receiveStream.add\,(event)$;
    if $event$ *is a channel close event* **then**
        $handle \leftarrow$ get the handle from the function parameter list;
        $stream \leftarrow stream\_trace.get\,(handle)$;
        if $stream$ *is not null* **then**
            $stream.closeStream.add\,(event)$;
            $stream \leftarrow True$;
return $stream\_trace$;

2.5 Stream Matching Algorithm: $streammatch()$

The communication identification algorithm aims at identifying all the communication of a concerned communication method from the dual-trace. The input of this algorithm is the two $stream\_trace$ from the dual-trace. The output of this algorithm is the communication list. Each communication recognized from the dual_trace contains two $stream$s. The channel of a communication defined in Section1.2 is not explicitly represented in the output but it was implicitly used in this algorithm.

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

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

To reduce the false positive error, the second level matching should be applied, which is also being named as transmitted data verification algorithm. On top of the endpoint identifiers matching, further data verification should be applied to make sure the matching is reliable. This verification crossly compare the sent and received data in both $stream$s in the first level matching. If the transmitted data in the $stream$s are considered to be identical, the matching is confirmed, otherwise it was a false positive error. However, we still can not exclude all the false positive errors, due to the data transmitted in two communication can be identical. Figure2.4 indicates the ineffective second

level matching scenario and the effective one.

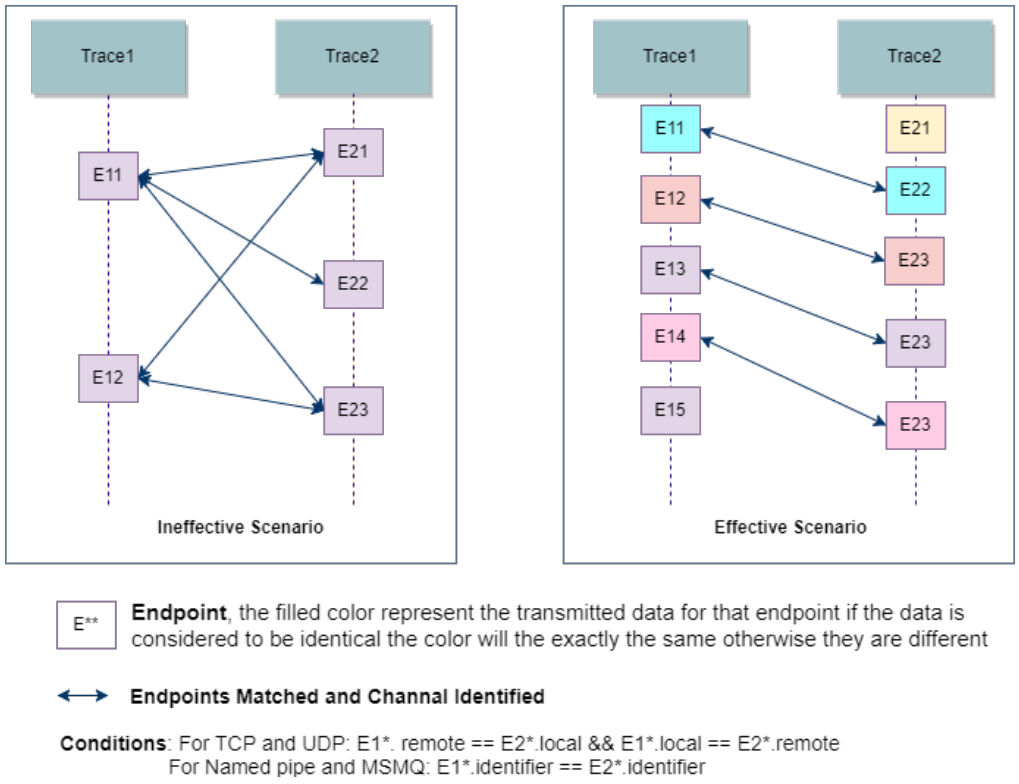

Figure 2.4: Second Level Matching Scenarios

The following subsections discuss the algorithms for these two level matching. In Section2.2, I elaborate the channel open process and the data transfer categories for the concerned communication methods. Based on the different channel opening process, two algorithms are developed for the communication identification, one is for Named Pipe and Message Queue, the other is for TCP and UDP. The inputs of the these two algorithms are the same, two $stream\_traces$ from the original dual_trace.

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

2.5.1 Stream Matching Algorithm for Named Pipe and Message Queue

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

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

Input: $stream\_trace_0, stream\_trace_1$
Output: $cos = \{co_y : 0 \leqslant y \leqslant Y\}$

```
1  cos ← Map⟨String, List⟨Communication⟩⟩;
2  for stream0 ∈ stream_trace_0 do
3      openEvent0 ← get the event from stream0.openStream, which should only contain
         one event;
4      channelId0 ← get the channel identifier from openEvent0.inputs;
5      for stream1 ∈ stream_trace_1 do
6          openEvent1 ← get the event from stream1.openStream, which should only
             contain one event;
7          channelId1 ← get the channel identifier from openEvent1.inputs;
8          if channelId0 == channelId1 then
9              DataVerified = dataVerify(stream0, stream1, outputdata). if
                 DataVerified == True then
10                 communication = New Communication();
11                 communication.stream0 = stream0;
12                 communication.stream1 = stream1;
13                 communication.dataMatch = outputdata; // The output from data
                     verification algorithm
14                 cos.add(communication);
15 return cos;
```

2.5.2 Stream Matching Algorithm for TCP and UDP

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

Algorithm 5: Stream Matching Algorithm for TCP and UDP

Input: $stream\_trace_0, stream\_trace_1$

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

$cos \leftarrow Map\langle String, List\langle Communication\rangle\rangle$;
for $stream0 \in stream\_trace_0$ **do**
  $socketEvent0 \leftarrow$ get the $socket()$ function call related event from $stream0.openStream$;
  $bindEvent0 \leftarrow$ get the $bind()$ function call related event from $stream0.openStream$;
  $connectEvent0 \leftarrow$ get the $connect()$ function call related event from $stream0.openStream$;
  for $stream1 \in stream\_trace_1$ **do**
    $socketEvent1 \leftarrow$ get the $socket()$ function call related event from $stream1.openStream$;
    $bindEvent1 \leftarrow$ get the $bind()$ function call related event from $stream1.openStream$;
    $connectEvent1 \leftarrow$ get the $connect()$ function call related event from $stream1.openStream$;
    if $socketEvent0! = null$ *AND* $socketEvent1! = null$ **then**
      if $bindEvent0! = null$ *AND* $connectEvent1 == null$ **then**
        $localServerAddr \leftarrow$ get the serverAddr parameter value from $bindEvent1.inputs$;
      else if $bindEvent1 == null$ *AND* $connectEvent0! = null$ **then**
        $remoteServerAddr \leftarrow$ get the serverAddr parameter value from $connectEvent1.inputs$;
      else
        Break the inner For loop;
      if $localServerAddr == remoteServerAddr$ **then**
        $DataVerified = dataVerify(stream0, stream1, outputdata)$.
        $communication = New\ \ Communication()$;
        $communication.stream0 = stream0$;
        $communication.stream1 = stream1$;
        $communication.dataMatch = outputdata$; `// The output from data verification algorithm`
        $cos.add(communication)$;
return cos;

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

As described in Section1.1.1, the data being received by one endpoint should always equal to or at least is sub string of the data being sent from the other endpoint in a communication for the reliable transmission methods, such as Named Pipe and TCP. So the data verification algorithm is in data union level. The send data union is retrieved by the concatenation of the input buffer content of the send events in the send stream of an endpoint. The receive data union is retrieved by the concatenation of the output buffer content of the receive events in the receive stream of the other endpoint. The input of this algorithm is the two $streams$ from two traces which are being matched in the first level.

Algorithm 6: Transmitted Verification by Data Union

Input: $stream0, stream1$

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

return *Indicator of if transmitted data union are considered to be identical* $send1 \leftarrow$ empty string;
$send2 \leftarrow$ empty string;
$recv1 \leftarrow$ empty string;
$recv2 \leftarrow$ empty string;
for $sendEvent \in stream0.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    $send0.append\,(sendmessage)$;
for $sendEvent \in stream1.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    $send1.append\,(sendmessage)$;
for $recvEvent \in stream0.receiveStream$ **do**
    $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
    $recv0.append\,(sendmessage)$;
for $recvEvent \in stream1.receiveStream$ **do**
    $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
    $recv1.append\,(sendmessage)$;
if $recv0$ *is substring of* $send1$ *AND* $recv1$ *is substring of* $send0$ **then**
    return True;
else
    return False;

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

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

Algorithm 7: Transmitted Verification by Data of Events

Input: $stream0, stream1$

Output: matched event list of two endpoints

return *Indicator of if transmitted data union are considered to be identical*
$sendPktNum0 \leftarrow stream0.sendStream.length$;
$sendPktNum1 \leftarrow stream1.sendStream.length$;
$recvPktNum0 \leftarrow 0$;
$recvPktNum1 \leftarrow 0$;
$eventMatchs \leftarrow List\langle EventMatch\rangle$;
for $sendEvent \in stream0.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    for $recvEvent \in stream1.receiveStream$ **do**
        $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
        if $sendmessage == recvmessage$ **then**
            $recvPktNum0++$;
            $stream1.receiveStream.remove\,(recvEvent)$;
            $eventMatch = NeweventMatch\,()$;
            $eventMatchs.add\,(eventMatch)$;
if $(sendPktNum0 - recvPktNum0) * 100/sendPktNum0 > threshold$ **then**
    return False;
for $sendEvent \in stream1.sendStream$ **do**
    $sendmessage \leftarrow$ get the input buffer content from the $sendEvent.inputs$;
    for $recvEvent \in stream0.receiveStream$ **do**
        $recvmessage \leftarrow$ get the output buffer content from the $recvEvent.outputs$;
        if $sendmessage == recvmessage$ **then**
            $recvPktNum1++$;
            $stream0.receiveStream.remove\,(recvEvent)$;
if $(sendPktNum1 - recvPktNum1) * 100/sendPktNum1 > threshold$ **then**
    return False;
return True;

2.6 Data Structures for Identified Communications

In the previous sections, I elaborate all the essential algorithms to identify the communications. The information of identified communications should be organized properly for the further presentation or visualization to the user. In this section, I define the output data structures to fulfil this requirement. There are totally two major data set. The first one is clustered as communications aligning the definition at Section1.2. The second one is clustered by endpoints in the traces. The reason to provide the second data set is due to the false positive errors of the channel identification. The identified endpoint lists of the traces provide more original data information. So with other assistant information and the access of this relatively original information of the dual-trace, the user has more flexibility to analysis the dual-trace. The data structures have been used in the algorithms implicitly.

Algorithm 8: Data Structure for Identified Communications

```
communications ← Map⟨String, List⟨Communication⟩⟩;
  stream_traces ← Map⟨String, List⟨Stream⟩⟩; struct {
    Stream stream0          // stream0 is from trace_0 of the dual-trace
    Stream stream1          // stream1 is from trace_1 of the dual-trace
    DataMatch dataMatch
} Communication
union {
    DataUnionMatch    unionMatch        // For data union verification
    List ⟨ EventMatch ⟩  eventMatchs     // For data event verification
} DataMatch
struct {
    String sData1                 // send data union of endpoint1
    String rData1   // receive data union of endpoint1,substring of sData2
    String sData2                 // send data union of endpoint2
    String rData2   // receive data union of endpoint2,substring of sData1
} DataUnionMatch
struct {
    Event      event1                 // event1 is from enpoint1
    Event      event2                 // event2 is from enpoint2
} EventMatch
struct {
    Int       handle
    List ⟨ Event ⟩  openStream
    List ⟨ Event ⟩  closeStream
    List ⟨ Event ⟩  sendStream
    List ⟨ Event ⟩  receiveStream
} Stream
struct {
    Int                       stratline
    Int                       endline
    Map ⟨ String, String ⟩  inputs
    Map ⟨ String, String ⟩  outputs
} Event
```

Chapter 3

Feature Prototype On Atlantis

In this section I describe the design of the feature prototype of communication identification from the dual_trace. This prototype is built on top of Atlantis' other features, such as "memory reconstruction", "function search" and "views synchronization". Atlantis is an assembly trace analysis environment. It provides many powerful and novel features to assist assembly level execution trace analysis.[1] This prototype implemented the algorithms described in Chapter2 as well as the user interfaces for the feature.

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

3.1 User Defined Function Set

As emphasized in Section2.2, the function set for each communication method can be different depends on the implementation solution of the method. Furthermore, there are so many communication methods in the real world and not all of them are being concerned by the user. Instead of using hard coded function sets, a configuration file in Json format is used for the users to define their concerned communication methods and the corresponding function set. This function sets will be the input for the communication identification. All concerned communication methods have its own function set. The identification features implemented in this prototype iterate all methods in the Json configuration file named "communicationMethods.json" and identify all communications of each method. This configuration includes the communication method, their function set for the communication events and the essential parameters of each function. A default

template is given for user reference, this default template is generated by Atlantis when it was launched and stored in the .tmp folder in the trace analysis project folder. Only Named pipe as a communication method is listed in this template. User can change this file to add or remove their concerned communication methods. The default template example can be find in Section4.3.

3.2 Parallel Editor View For Dual_Trace

The dual_trace consist of two execution traces which are interacting with each other. To present them in the same view makes the analysis for the user much easier. The strategy to open parallel editor view is that open one trace as the normal one and the other as the dual_trace of the current opened one. A new menu option in the project navigation view are created for opening the second trace as the dual_trace for the current active trace editor. The implementation of the parallel editor take the advantage of the existing SWT of Eclipse plug-in development. The detail of the implementation can be found in Section4.4. Figure3.1 shows this menu option and Figure4.4.

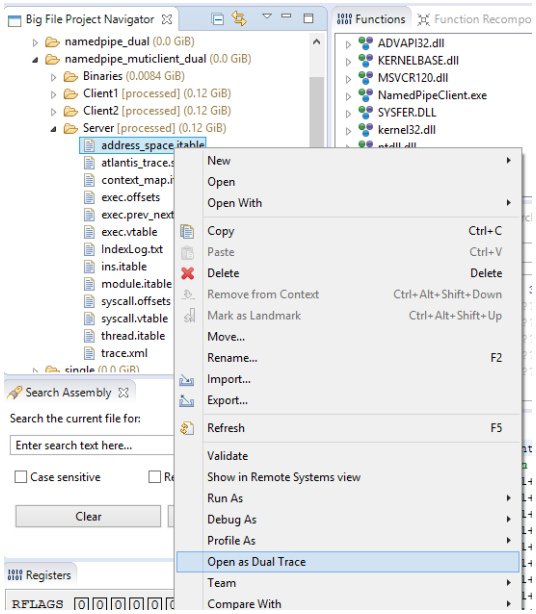

Figure 3.1: Menu Item for opening Dual_trace

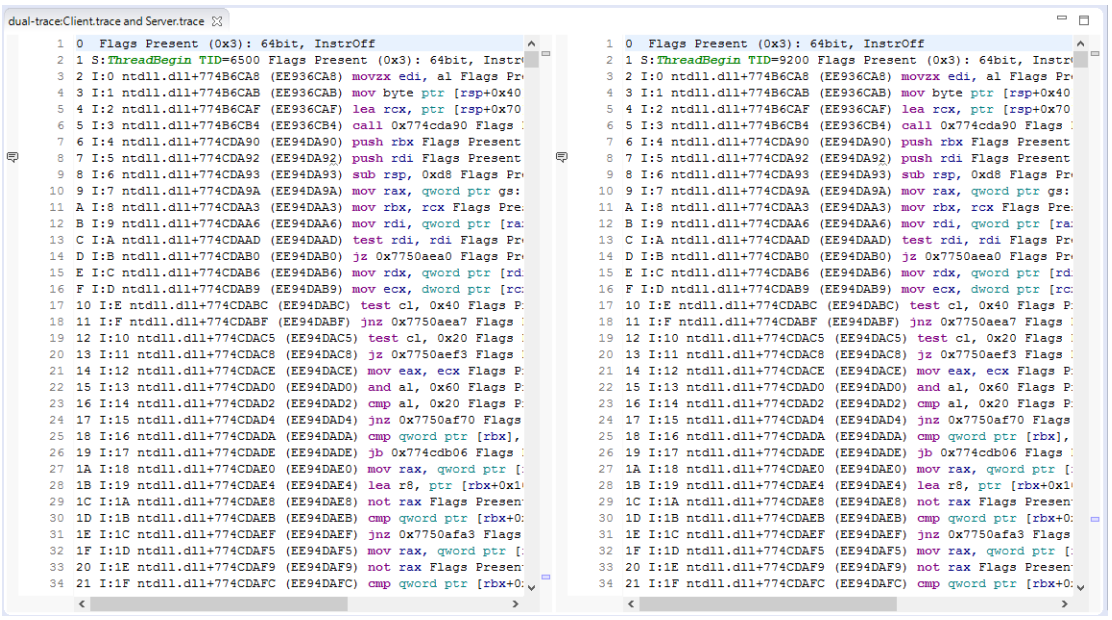

Figure 3.2: Parallel Editor View

3.3 Identification Features

The called functions' name can be inspected by search of the symbolic name in the executable binary or any DLLs which used by the program at the time when it is traced. This functionality exists in the current Atlantis. By importing the DLLs and execution executable binary, Atlantis can list all called functions for the users in the Functions view. From this list, users can chose the interested functions and generate their interested communication type. In Figure3.3 there is an action item "Add to Communication type" in the right click menu of the function entry. Figure 3.4 shows the dialogue for entering the information for the adding function. As this figure shows, users can get the existing communication type list in the drop down menu. They can choose to add the current function to an exist communication type or they can add it to a new communication type by entering a new name. For the channel create/open function, the register holding the address of channel's name as input and the register holding the handle identification of the channel as output are required. For the send/receive function, the register holding the address of the send/receiver buffer, the register holding the length of the sending/receiving message and the register holding the channel's identification are required. As there are 4 functions for each communication type users have to repeat this add function to communication type action for 4 times to generate one communication type.

3.3.1 Stream Identification Feature

3.3.2 Communication Identification Feature

3.4 Identification Result View and Result Navigation

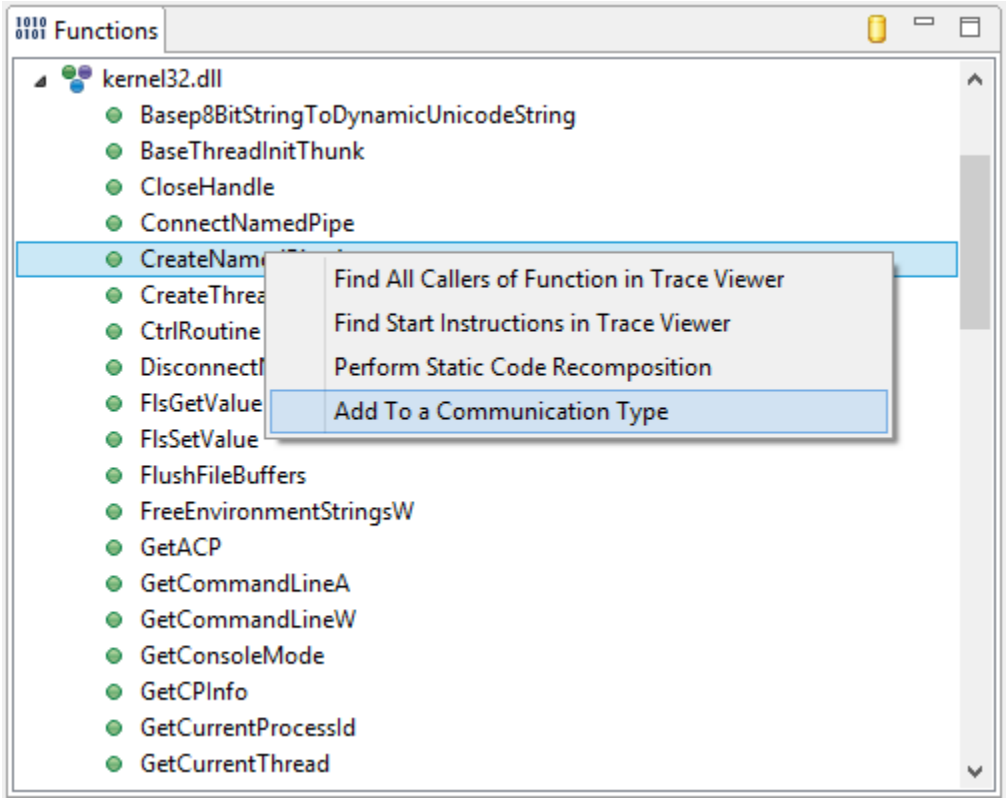

Figure 3.3: Add function to a Communication type from Functions View

The right click menu of an entry in the search result list has two action items: Go To Line of Message Sender and Go To Line of Message Receiver. Both of the action items allow users to navigate to the trace Instruction view. When the user click on these items, it will navigate to the corresponding trace sender or receiver trace instruction view. Meanwhile the memory view jumps to the target address of the message buffer, and the memory state is reconstructed so that the message content in that buffer will be shown in the memory view.

A new view named Communication Types view is for the user defined communication types. All user defined communication type are stored in the .xml file and listed in communication type view when it's opened as shown in Figure 3.5. User can change the name of a communication type, remove an existing communication type or searching of the match message occurrences of

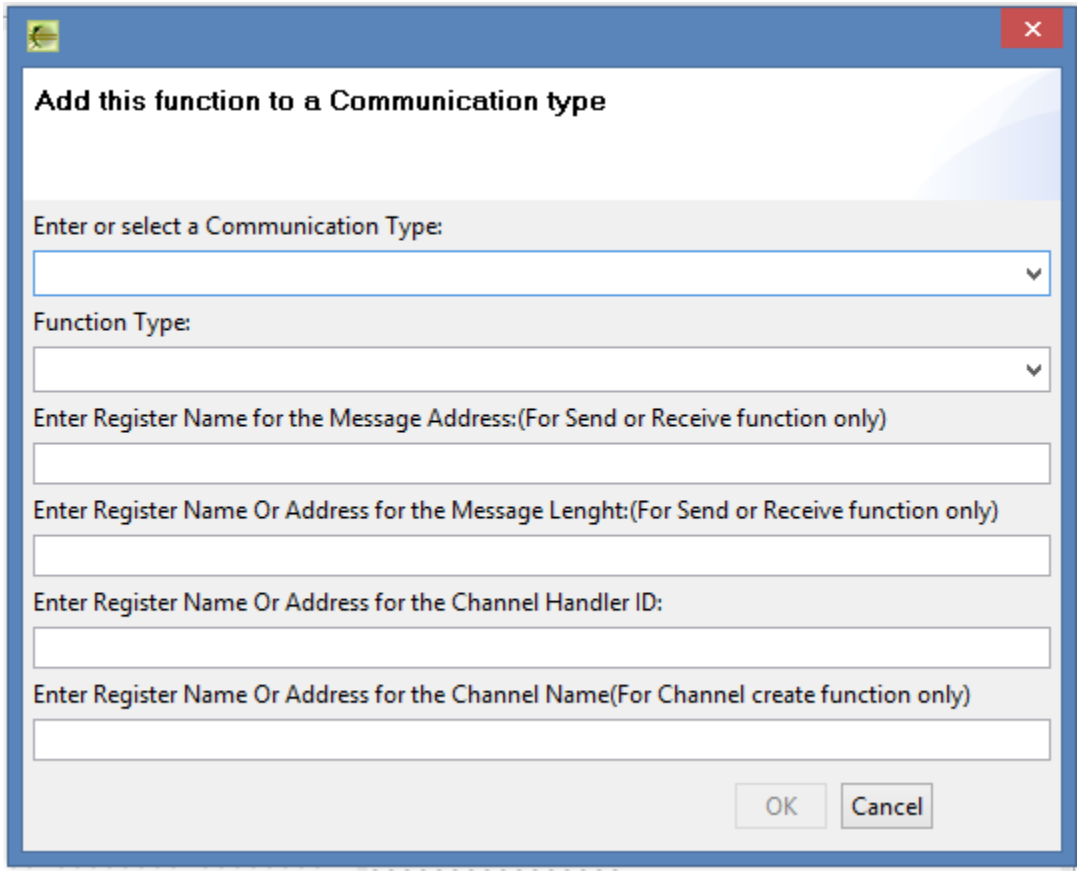

Figure 3.4: Dialog to input information for a function adding to a communication type

selected communication type by selecting action item in the right click menu of an communication type entry. The matched messages are listed in the result window of the view. By clicking the entry of the search result, user can navigate to it's sender or receiver's corresponding instruction line as shown in Figure3.6. Message content in the memory view will be shown as well.

The communication event consists of the send message event in the sender side and receive message event in the receiver side. The communication event searching algorithm can be divided into three main steps: 1. search all channel create/open event in the sender and receiver side, save the handle id and corresponding channel name. 2. Search all message send and receive event in sender and receiver sides. 3. Matching the send/receive messages pair based on the channel names and message contents.

In this step the algorithm is supposed to search all the open channel both in the sender and receiver side. The found created channels are recorded in a map. The key of the map is the handler id of the channel and the value is the channel name. A channel in the sender and receiver sides will have different handler id but same channel name.

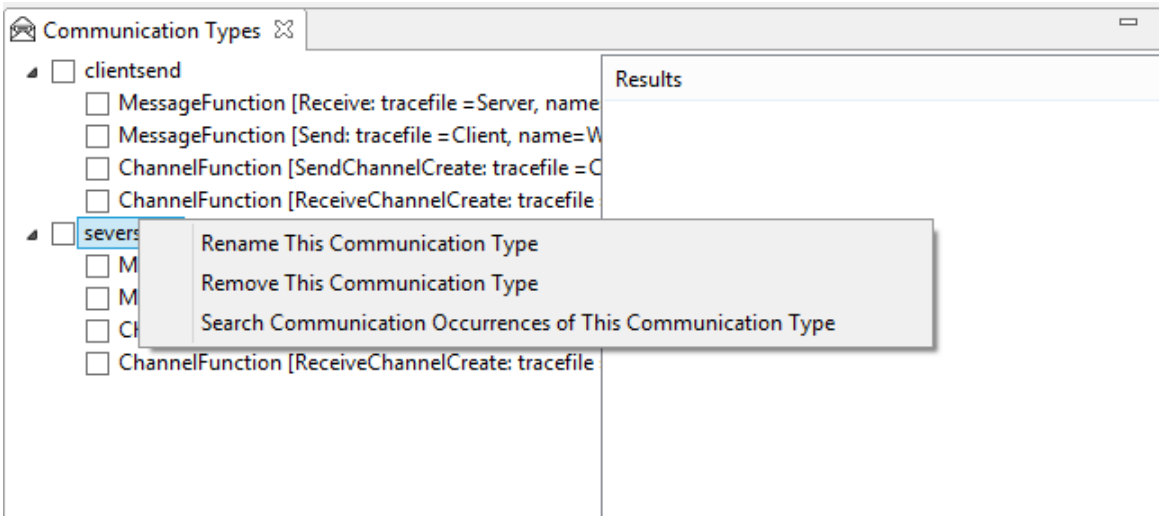

Figure 3.5: New View: Communication Type View

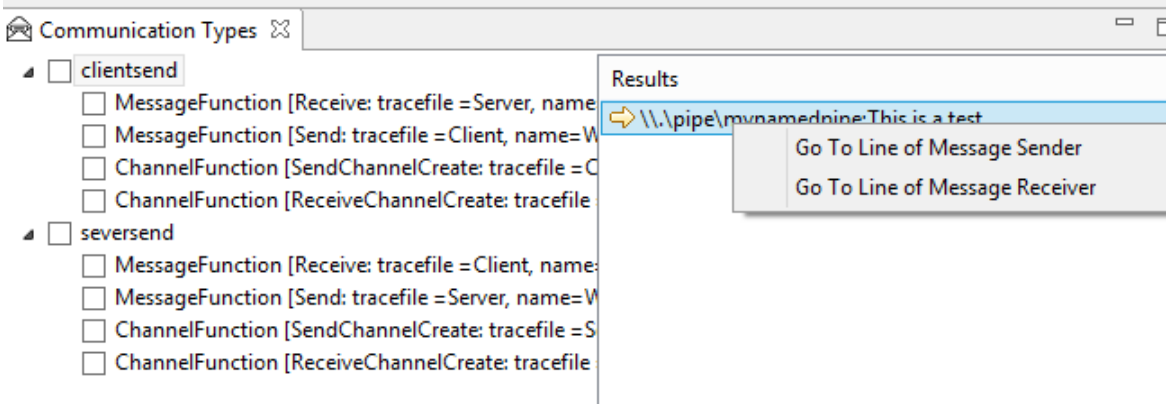

Figure 3.6: Right Click menu to navigate to send and receive event in the traces

All send message and receive message function calls will be found out in the trace. When a send function hit, the memory state of the hit instruction line will be reconstructed, and the message content can be get from the memory with the send message buffer address. When a receive function hit, the return line of that function is needed for getting the message content. The memory state of the function return line is reconstructed and the message content can be get from the reconstructed memory state with the receive message buffer address.

After the created channel and send/receive message are found out in the sender and receiver side, a matching algorithm is used to match the send/receive message pairs.

The matching event is stored in cache when the tool is running. Only the most recent search result is cached currently. If users need the previous result, they need to apply the search again. The matching Event consist of two sub-events, one is message send event while the other is message

receive event. Both of these two sub-events are object of BfvFileMessageMatch. BfvFileMessageMatch is an Java class extends org.eclipse.search.internal.ui.text.FileMatch. FileMatch class containing the information needed to navigate to the trace file editor. In order to show the corresponding send/receive message in the memory view, the target memory address storing the message content is set in BfvFileMessageMatch. Two more elements: message and channel name are also set in BfvFileMessageMatch which are listed in the search result.

Chapter 4

Additional Information

4.1 Terminology

Endpoint:
An instance in a program at which a stream of data are sent or received (or both). It usually is identified by the handle of a specific communication method in the program. Such as a socket handle of TCP or UDP or a file handle of the named piped channel.
Channel:
A conduit connected two endpoints through which data can be sent and received
Channel open event:
Operation to create and connect an endpoint to a specific channel
Channel close event:
Operation to disconnect and delete the endpoint from the channel.
Send event:
Operation to send a trunk of data from one endpoint to the other through the channel.
Receive event:
Operation to receive a trunk of data at one endpoint from the other through the channel.
Channel open stream:
A set of all channel open events regarding to a specific endpoint.
Channel close stream:
A set of all channel close events regarding to a specific endpoint.
Send stream:
A set of all send events regarding to a specific endpoint.
Receive stream:

A set of all receive events regarding to a specific endpoint.

Stream:

A stream consist of a channel open stream, a channel close stream, a send stream and a receive stream. All of these streams regard to the same endpoint.

4.2 Microsoft* x64 Calling Convention for C/C++

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

4.3 Example of Configuration File for Communication Methods' Function Set

```
[
  {
    "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": "CreateFileA",
    "createHandle": true,
    "type": "open"
  },
       {
    "retrunValReg": {
      "name": "RAX",
      "valueOrAddress": value
    },
    "valueInputReg": {
      "name": "RCX",
      "valueOrAddress": value
    },
    "memoryInputReg": {
      "name": "RDX",
      "valueOrAddress": address
    },
    "memoryInputLenReg": {
```

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

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

4.4 Open View with Two Parallel Editors programmatically

4.4.1 The Editor Area Split Handler

```
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;
        }
}
```

4.4.2 Get the Active 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);
```

Bibliography

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

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

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

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