SMT: A System Monitoring Tool for DCE

Ganesh Jayadevan Ramesh Narayan Charlie Richardson Neal Krawetz Anil Gurijala
Anish Karmarkar Paul Brutch Kyle Walzel Vance Swaggerty Ganesha Beedubail
Sridhar Muppidi Sunil Santha Willis Marti Udo Pooch

February 28, 1995

Abstract. *Understanding interactions among various components in a distributed system is very important for system administrators and application developers. The System Monitoring Tool*[1] *(SMT) was developed to help in this regard for the OSF/DCE environment. SMT consists of two components: a remote monitoring agent to capture packets, and a management workstation which analyzes and groups them. The interactions can be viewed at three different levels, from raw packets to complete transactions. This helps the user to quickly analyze the interactions in a remote procedure call.*

Key Words. Network Monitoring, Distributed Computing Environment, Remote Procedure Call, Client Server Model.

1 Introduction

The advancement of computer network technology has made distributed processing feasible and popular. Various models and platforms are being developed to make the development and implementation of distributed applications easier. OSF's Distributed Computing Environment (DCE) is one of the most popular platforms which support distributed applications. DCE is a software layer placed between a distributed application and operating system/network services. It provides application programmers with the necessary tools to assist in application program development. It also provides a set of integrated services which allow distributed applications to interact with a collection of heterogeneous computers, operating systems and networks as if they were a single system. DCE is based on the client/server model of computing. In this model, the client and server are different processes, probably running on different machines. Servers provide services which are requested by clients. The most important mechanism used to implement the client server model is Remote Procedure Call (RPC). This mechanism allows a client to make calls to remote procedures running on server machines. The implementation of RPC makes remote operation as transparent as possible. Remote procedures are invoked in the same manner as local procedures.

In DCE, the interaction among applications and system services, like in any other distributed system, is complex. Therefore, DCE system managers and application developers may need to study

1. This project was supported by IBM, Austin under contract number 32525–42640–CS.

the network traffic in order to trouble shoot distributed applications and to better understand the interactions among various components. One of the most popular ways of trouble shooting is to capture packets on the network and analyze them. There are a number of existing tools available for this function, but they are not helpful in analyzing DCE RPC packets. They are limited to monitoring a single packet and do not show the relationship between various packets of an RPC. The System Monitoring Tool (SMT) design offers two key advantages: it allows simultaneous monitoring of multiple network segments, and provides protocol–aware decoding and display of RPC activity. In essence, it allows RPC traffic to be viewed at different levels of granularity – starting from the transaction level to the exchange primitive level and finally to the packet level where one could see the actual bytes in a packet.

In Sections 2, 3 and 4, a brief introduction to DCE, SNMP and RMON MIB is given for the benefit of reader. For further details, the reader is advised to read the references given. In sections 5 and 6, the rationale and overview of SMT are given. From Section 7 to 11, each component of SMT is explained in detailed. Section 12 gives the current status and Section 13 gives the scope to future work.

2 Interprocess Communication in DCE

DCE is a client–server environment where servers provide services requested by clients. To request a service, a client invokes a remote procedure call. The DCE RPC runtime (a group of library routines) takes care of the underlying communication between the client and server. During an RPC call, the client and server–side runtime mechanisms handle client server mapping (with the help of *cell directory service* (CDS)), establish communication over an appropriate protocol, pass call data between the client and server, and handle communication errors. The RPC runtime supports both connection–oriented and connectionless protocols – TCP and UDP, respectively. The communication is done by RPC protocol data units (PDUs). Connection–oriented and connectionless protocols have different types of PDUs. The information and structure of each PDU is given in Chapter 12 of [AES]. Each remote procedure call involves the exchange of more than one PDU between the client and server. For example, the following PDUs are involved in a normal RPC:

1. The client sends a REQUEST PDU to the Cell Directory Service (CDS) requesting the address of a server.
2. The CDS responds by sending a RESPONSE PDU which contains the address of the server.
3. The client sends a REQUEST PDU (which may contain parameters of an RPC) to the server.
4. The server sends a RESPONSE PDU containing the result.

The completeness and correctness of an RPC event depends on the correctness of these PDUs.

DCE nodes may exist on any of the common types of networks – token ring, Ethernet, FDDI, and so forth. This must be considered when decoding DCE packets, as well as the network and transport layer protocols the packet is following. An example RPC DCE packet on an Ethernet network is shown in Figure 1.

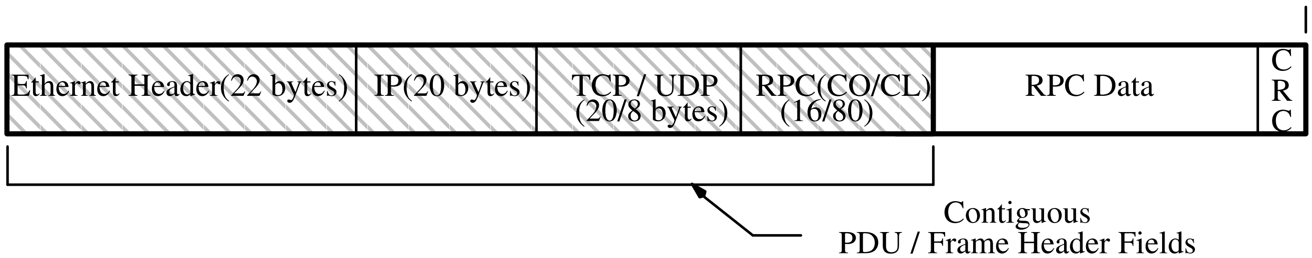

Figure 1. Protocol Headers of RPC Ethernet Packets.

3 SNMP Protocol

Simple Network Management Protocol, popularly known as SNMP, is a standard developed to provide a basic and easily implemented network management tool for TCP/IP based environments.

The model of network management that is used for TCP/IP network management includes the following key elements[Stallings]:

Management Station

Management Agent

Management Information Base

Network Management Protocol

A *Management Station* serves as the interface between the human network manager and network management system. *Management Agents* (or Remote Monitoring Agents) are platforms such as hosts, bridges, and routers that are equipped with SNMP so that they can be managed from the manager.

The resources in the network are represented by *objects*. Each object is a data variable that represents one aspect of the managed agent. The collection of objects is referred to as the *Management Information Base (MIB).* The MIB functions as a collection of access points at the agent for the management station. These objects are standardized across systems of a particular class. The structure of management information (SMI) [RFC 1155] defines the general framework within which an MIB can be defined and constructed. Associated with each type of object in an MIB is an identifier of the ASN.1 type OBJECT IDENTIFIER. The identifier serves to name the object. In addition, because the value associated with the type OBJECT IDENTIFIER is hierarchical, the naming convention also serves to identify the structure of object types. An object identifier is a unique identifier for a particular object type. Its value consists of a sequence of integers. The set of defined objects has a tree structure, with the root of the tree being the object referring to the ASN.1 standard.

The management station and agents communicate with each other using SNMP messages, which include the following key capabilities:

Get – Enables the management station to retrieve the value of objects at the agent.

Set – Enables the management station to set the value of objects at the agent.

Trap – Enables an agent to notify the management station of significant events.

A management station performs the monitoring function by retrieving the value of MIB objects. A management station can cause an action to take place at an agent by modifying the value of specific variables.

4 RMON MIB

Internet RFC 1271 defines a remote–monitoring MIB that supplements MIB–II and provides the network manger with vital information about the network. This RFC lists the design goals for a remote monitoring agent (RMON). These goals are to support off–line operation, multiple managers, preemptive monitoring, and problem detection and reporting, as well as to provide value–added data.

The two main operations that a management station performs on an RMON agent are *configuration* and *invocation*. During configuration, the agent is configured for data collection. The configuration dictates the type and form of data collected. The MIB is organized into various functional groups. Each group consists of one or more control tables, and one or more data tables. A control table, which is typically read–write, contains parameters that describe the data in the data table, which is typically read–only. At configuration time, the management station sets the appropriate control parameters to configure the remote monitor to collect the desired data. These parameters are set by adding a new row to the control table, or by modifying an existing row. As information is collected according to the parameter settings of a control row, it is stored in rows of the corresponding data table. To modify any parameters in a control table, it is necessary to first invalidate the control entry (row). This causes the deletion of that row and all associated rows in data tables. The management station can then create a new control row with the modified parameters. The same mechanism is used to simply disable a particular data collection function. When a row of a control table is deleted, the associated data table rows are deleted, and the resources used by those rows are reclaimed.

For *invocation*, SNMP protocol provides no specific mechanism for issuing a command to an agent to perform an action. SNMP's only capabilities are to read object values, set object values, and set object values within an MIB view. However, it is possible to use the SNMP set operation to issue a command. An object can be used to represent a command, so that a specific action is taken if the object is set to a specific value. A number of such objects are included in the RMON MIB. In general, these objects represent states, and an action is performed if the management station changes that state (by changing the value of the object). A request to set an object to its current value does not cause an action to be performed.

The RMON MIB is divided into nine functional groups as shown in Figure 2. Of these, we are interested in only three: *filter*, *packet capture* and *event*.

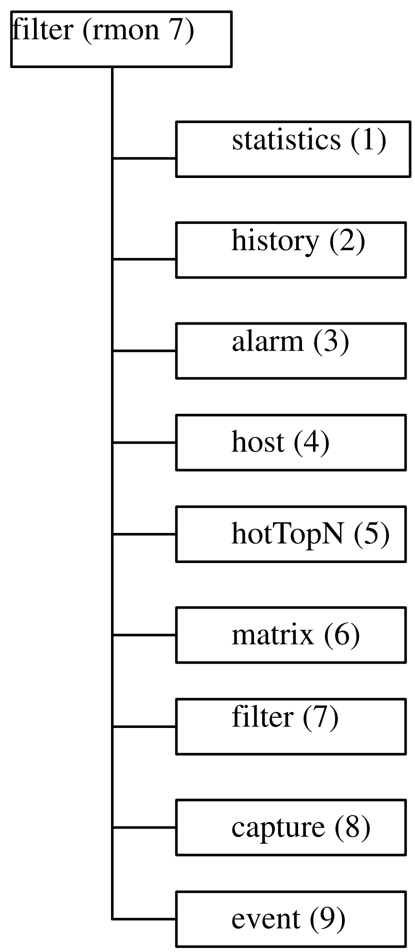

Figure 2. Functional Groups of RMON MIB.

The *filter* group provides a means by which a management station can instruct a monitor to observe selected packets on a particular interface. The basic building blocks defined in this group are two kinds of filter: a data filter and a status filter. The data filter allows the monitor to screen observed packets on the basis of a bit pattern which a portion of the packet matches (or fails to match). The status filter allows the monitor to screen observed packets on the basis of their status (valid, CRC error, and so forth). These filters can be combined using logical AND and OR operations to form a complete test to be applied to incoming packets. The stream of packets which pass the test are referred to as a channel, and a count of such packets is maintained. In addition, the channel can be configured to generate an event, defined in the *event* group, such that when a packet passes through a channel it can be captured if that mechanism is defined in the capture group. The logic defined for a single channel is quite complex. This gives the user enormous flexibility in defining the stream of packets to be counted.

The *packet capture* group can be used to set up a buffering scheme for capturing packets from one of the channels in the filter group. It consists of two tables: `bufferControlTable`, which specifies the details of the buffering function, and `captureBufferTable`, which buffers the data. Each row in the `bufferControlTable` defines one buffer that is used to capture and store packets from one channel. The data table, `captureBufferTable`, contains one row for each packet captured.

The *event* group supports the definition of events. An event is triggered by a condition located elsewhere in the MIB, and an event can trigger an action defined elsewhere in the MIB. An event may also cause information to be logged by this group, and may cause an SNMP trap message to be issued.

5 SMT Rationale and Goals

Existing tools are not sufficient for analyzing DCE RPC packets. They are limited to monitoring a single packet and do not show the relationship between various packets of an RPC. CITI at Michigan did some work in this regard [HOW93]. Their *Sniffer* captures DCE RPC packets and analyzes them. However, it has some drawbacks. One is that the DCE IDL needs to be modified in order to analyze packets by the Sniffer. Another disadvantage is that the Sniffer has to be run on each host wherever capturing and analyzing are required. A third drawback is that the Sniffer performs unmarshalling(un–packing of abstract data types, dereferencing pointers etc.) of packets. All of these drawbacks are overcome by SMT.

For SMT, no modification of any DCE components are necessary. Only one management host running SMT is sufficient. However, at least one RMON is needed on each network segment containing DCE nodes.

The goals and focus of the SMT project were as follows:

1. Develop a “standalone” software tool, independent of the running status of DCE on the network which provides network managers, system managers, and

programmers the ability to examine the state of RPC protocol transactions within a DCE cell network (consisting of multiple machines and multiple subnetworks) from a single workstation host located on any subnetwork.

2. Develop a set of functions in the implementation of this tool which may be used as ”building blocks” for incorporation of SMT functionality into both other tools and other specific network/system management schemes.

3. The tool should be generic enough to be portable to different operating environments, as well as commonly used protocols.

6 SMT System Overview

SMT consists of one program running on a host workstation, and one or more RMONs connected to network segments containing DCE machines. These RMONs must conform to the RMON MIB Internet standard, and may be either dedicated hardware, or programs running on workstations or PCs. The SMT remote monitoring software runs on the host workstation, and controls the RMONs via SNMP (Simple Network Management Protocol).

The SMT components in operation allow the independent monitoring of RPC protocol transactions on the monitored network segments in order to verify DCE inter–machine communications and proper network software and hardware operations. The SMT host workstation program has a user interface and a modular design, which can be upgraded or extended with new functions and features. Besides doing basic decoding of individual packets, SMT also identifies all packets related to a transaction, and which transaction primitives (e.g., INVOKE, RESPONSE, CANCEL) have occurred.

SMT is designed to provide verification of proper network and DCE functioning, as well as to assist in software development and debugging. An example DCE network configuration consisting of DCE client and server machines, network segments, bridges, and the SMT components is shown in Figure 3.

This document follows the naming convention presented in [ROE91]. The SMT is composed of two major components: System Monitoring Tool software component, and RMON Agent hardware component, as shown in Table 1. Each such module is called a Computer System Configuration Item (CSCI). The SMT software component is further broken down into functional units called Computer Software Components (CSC) as shown in Table 2. Each CSC consists of Computer Software Units (CSU), the smallest unit identified in [ROE91]. A characteristic feature of a CSU is that it corresponds to a unit of compilation (in terms of software).

Table 1. Summary of SMT Components		
Section	Description	ID
CSCI 1	Remote Monitoring Agent hardware component, implementing SNMP and RMON MIB (RFC 1271) capability	CSCI RMON
CSCI 2	System Monitoring tool software component. Refer to Table 2.	CSCI SMT

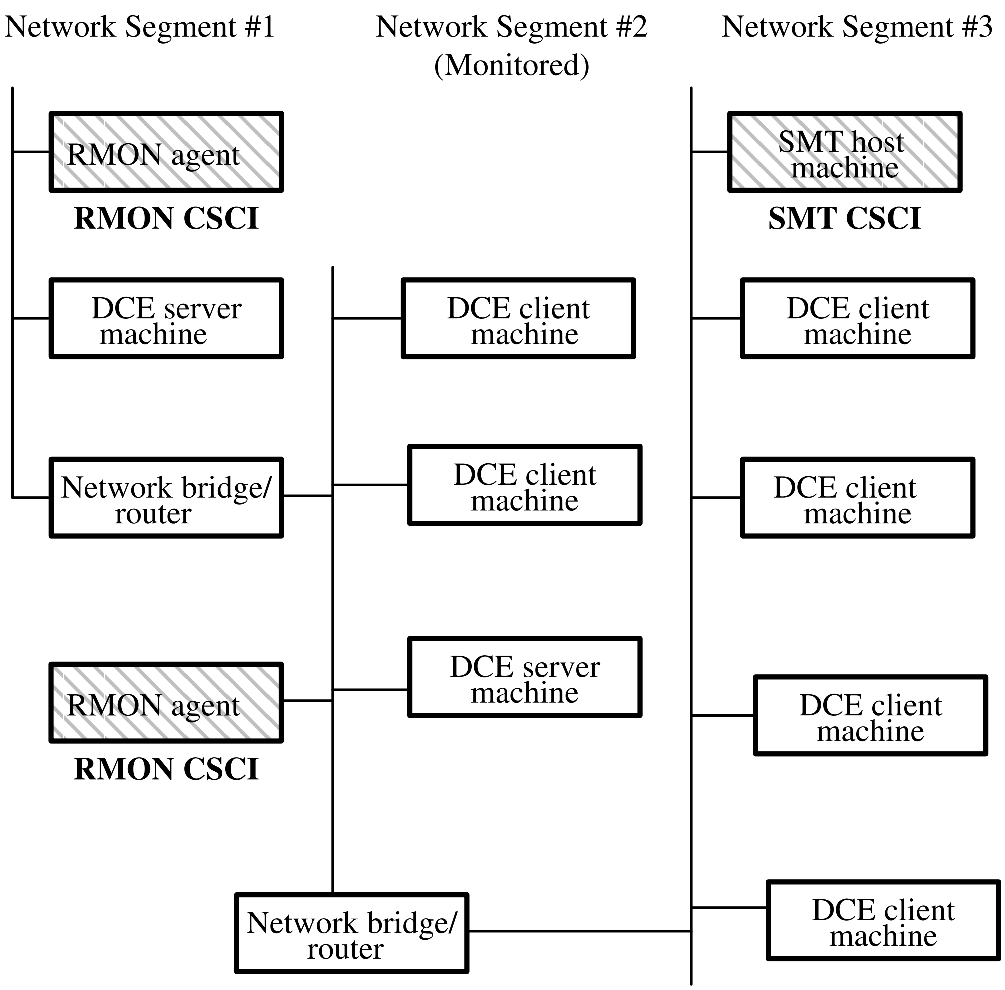

Figure 3. SMT Component Network Locations.

Table 2. Summary of SMT Host Workstation Software CSC Components		
Section	Description	ID

CSC 1	Remote Packet Capture software – performs communications with remote monitoring agent (CSCI 1). Collects and logs captured RPC packet header data for later analysis.	RPCAP CSC
CSC 2	Packet Processor software – performs RPC transaction analysis.	PP CSC
CSC 3	Graphical User Interface software – performs user interface, monitoring, and control functions.	GUI CSC
CSC 4	Communications software component – Provides interprocess communications between RPCAP, PP, and GUI CSC's.	COMM CSC

The main functions of SMT are to capture DCE RPC packets on the network, analyze, and group them to get proper RPC semantics. The capturing of packets is done by the RPCAP CSC of SMT, which controls the RMON agents. The analysis and grouping of individual packets is done by the PP CSC of SMT.

7 SMT RMON Agent

The SMT RMON agent is a network–connected hardware component which supports MIB according to SNMP standards (as discussed in Section 4). This may be a dedicated instrument, or a PC or workstation running appropriate software. Typically, one RMON agent is connected to each network segment. In our lab, we used two RMON agents – an ethermeter and a PC running OS/2 RMON software. The software used was Beholder – The Next Generation[BTNG], an RMON–compliant monitoring program. This OS/2 application turns a PC with a dedicated network card into a remote Ethernet monitor for use with SMT.

The SMT CSCI software operating on the SMT host workstation controls the remote monitoring capabilities of the RMON agent(s). SMT sends SNMP messages to each RMON agent to collect and store RPC packets (or fragments of packets, particularly initial headers) and forward these packets (or packet fragments) to SMT upon request. This can be done over either the general network, or a dedicated communications link to SMT. Typically, the SMT software will poll the RMON agent(s) periodically to collect only those packets captured since the last poll.

The main functions of an RMON agent are to inform the SMT host workstation about the type of interface on which it is connected (such as Ethernet), and to capture and forward packets to the SMT host workstation. Only *filter* and *packet capture* groups are used for this purpose. The *filter* group of the RMON agent is set to capture only DCE packets. Basically, the agent captures only those packets which match all of the following fields.

1. The field identifying the TCP/IP protocol.
2. The field identifying the connection type (TCP or UDP)
3. The field identifying the DCE RPC.

The number of bytes to be captured and forwarded is set by the RPCAP CSC of SMT. The communication between the SMT management station and agent is done using SNMP messages.

8 Remote Packet Capture

The Remote Packet Capture or RPCAP CSC is an SMT process initiated by the GUI CSC. RPCAP runs as a separate process on the SMT host workstation and is responsible for communication between the manager host and RMON agents. RPCAP uses SNMP messages to communicate with RMON agent(s). After collecting the packet fragments and timestamp information from the RMON agent(s), the RPCAP CSC then stores this data in a log file on the host workstation for later analysis by the PP CSC. The logical flow diagram of RPCAP is shown in Figures 4 and 5.

The RPCAP CSC performs the following functions:

1. Receive commands from the Graphical User Interface (GUI) CSC to configure an RMON agent, and to start and stop packet capturing by an RMON agent.
2. Periodically poll RMON agent(s) by using SNMP messages, when packet capture is active. Request and receive captured UDP datagram fragments from the RMON agent(s).
3. Store captured RPC packet headers in a log file to allow later analysis by the PP CSC.

When RPCAP is created by the GUI, it performs the following functions during initialization:

1. Build the MIB file name tree from the MIB text file.
2. Initialize (opens) the COMM CSC message queue shared with PP and GUI.
3. Send a READY message to the GUI to confirm process initialization.
4. Call a blocking message receive function to wait for a GUI command.

Normal operation of RPCAP involves an infinite loop. The RPCAP process performs a message receive call, which blocks the process from execution. The call will either return normally if there is a message for RPCAP in the message queue, or will terminate prematurely with an error, if the message receive call is interrupted by a signal. If the SIGALRM signal has been received and capture is active, RPCAP will poll the RMON agent(s) and fetch any captured RPC packet header data. Then it resets the alarm signal timer (via the alarm() call) for the next polling interval (normally one to ten seconds) and returns the message receive call. This approach ensures that excessive CPU cycles are not consumed in a ”spin wait” for either the availability of the message or the arrival of the polling alarm.

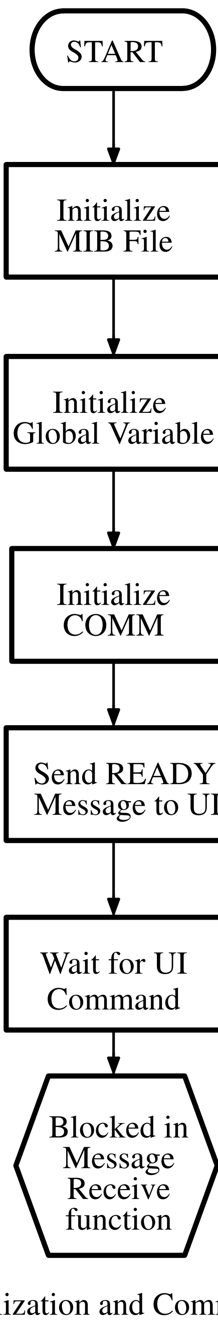

Figure 4. RPCAP Initialization and Command/Polling.

The commands received by RPCAP CSC from the GUI CSC are CONFIGURATION, START CAPTURING and STOP CAPTURING. This CSC decodes the commands and performs appropriate actions. When a configuration command is received, this CSC initializes the specified RMON agent by setting up a session with that RMON agent. The filters, channels and other variables are set in the RMON agent and made ready to capture packets.

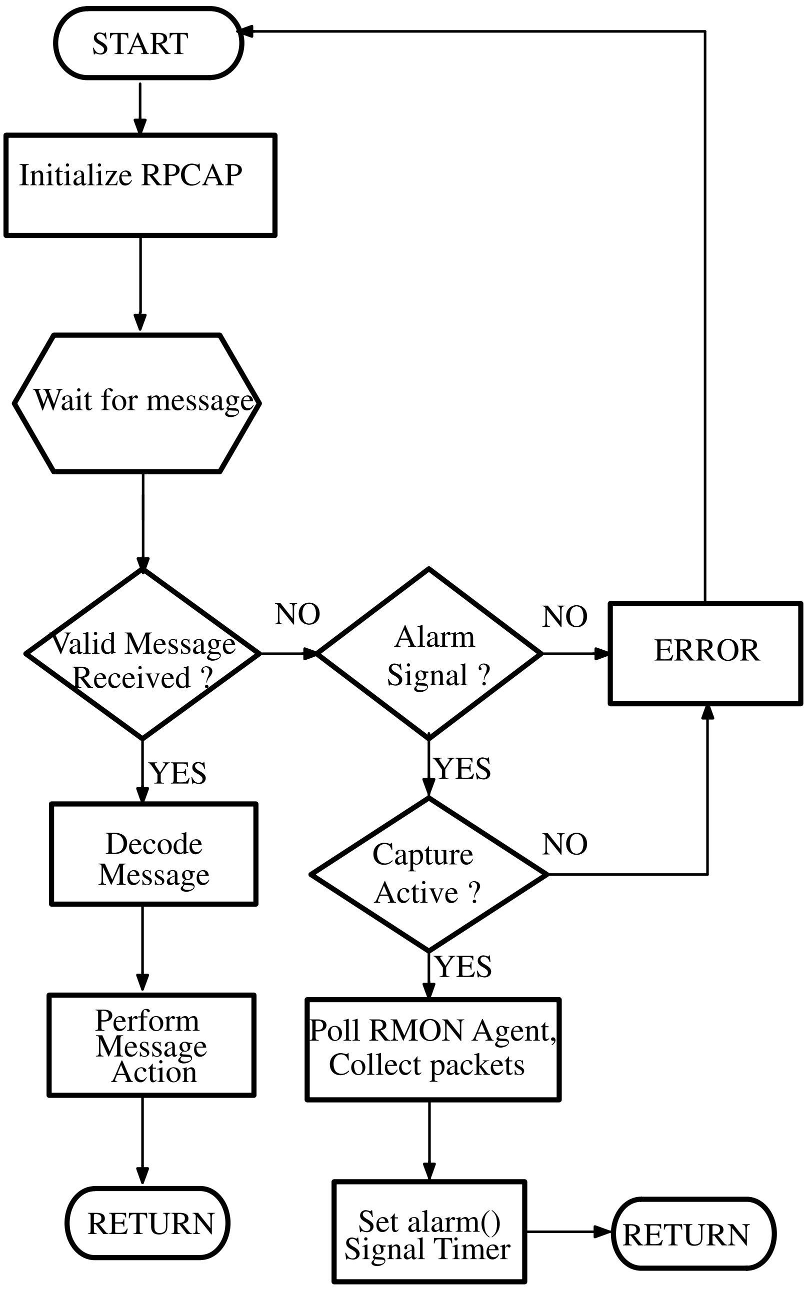

Figure 5. Command Polling and Execution.

RPCAP does the following operations on the objects of the filter and capture groups while configuring an RMON Agent:

1. Sets `channelStatus` table valid in filter group.

2. Initializes rows of `bufferControlEntry` table in the capture group to appropriate values. These values are setting owner, packet capture size, packet forward size, offset from starting bit, size of the buffer to be allocated for captured packets, and so forth. Note that it is assumed (for now) in SMT that the Options/ Padding fields of IP and TCP headers are assumed to be empty. This means that the IP and TCP headers are both assumed to be a fixed length of 20 bytes.

3. Initializes rows in `channelEntry` and `filterEntry` tables to capture required packets.

To start capturing packets, RPCAP sets `BufferControlStatus` row as valid and `Channel-DataControl` as "ON". To stop capturing packets, RPCAP sets `ChannelDataControl` as "OFF". While capturing packets, RPCAP polls an agent at regular intervals. It sends *GetRequest* and *GetNextRequest* messages to the agent to read values in the `captureBufferPacketData` and `captureBufferPacketTime` columns of the `captureBufferEntry` table of the capture group.

9 Packet Processor

The Packet Processor analyzes files logged by RPCAP CSC. The PP analyzes packet header streams and reconstructs complete or partial RPC protocol transactions. Analyzed transaction streams are maintained in files for permanent storage and display access by the GUI.

The PP CSC runs as a single separate process under SMT. It communicates with the GUI CSC using the COMM CSC message queue. The purpose of the PP is to logically group an arbitrary collection of RPC packets into transactions. Both connection–oriented and connectionless RPC calls have well–defined RPC protocol states. Grouping the RPC PDUs according to the state of the RPC transaction allows for a higher level semantic view of ongoing transactions between two distributed programs in DCE.

A connection–oriented (CO) RPC makes use of TCP and connectionless (CL) RPC UDP protocols. Thus, CO and CL RPCs differ in the transport protocol used to send packets between a client and server. The determination of whether a packet in a log file is CO or CL is done by reading the Protocol field in the IP header of the packet. If this value is 6, then the protocol is TCP, and the packet belongs to a CO transaction. And, if the value is 17, then the protocol is UDP, and the packet belongs to a CL transaction.

9.1 RPC Transaction Definition

The interaction between client and server can be viewed at three different levels. The highest level is the *transaction* level. A *transaction* is defined as a single complete remote procedure call between a client and server by means of the RPC protocol. One remote procedure call transaction may result in the exchange of many PDUs between the client and server. All the RPC PDUs exchanged between the client and server during the interaction belong to that transaction. The lowest level is *packet* level. At this level, the user can see the packets transmitted between the client and server due to an RPC call. Each packet consist of at most one RPC PDU header. The RMON agent captures only those packets which have an RPC PDU header. Each RPC transaction may result in more than one PDU. Therefore, an intermediate level is created by grouping certain types of PDUs. This level is called the *Exchange Primitive* or *XP* level. The Exchange Primitives are explained further in the next section.

9.2 Exchange Primitive Definition

Exchange Primitives (XP) are different phases of an RPC transaction during the client–server interaction. We have developed a set of Exchange Primitives, whose concept is borrowed from the AES DCE service primitives [AES].

According to the AES DCE, "the RPC service primitives provide an abstract implementation of the specified RPC call semantics and serve to map the specified semantics to the specified protocol machines." The following is a list of the service primitives and their definitions [AES]:

> **Invoke** – used to invoke an RPC. This is a service user initiated service primitive.
>
> **Result** – used to return the input/output parameters at the end of a normal execution of the invoked RPC. This is a server–user initiated service primitive.
>
> **Cancel** – used to cancel an outstanding RPC.
>
> **Error** – used by the server manager routine to indicate an error in response to a previous Invoke indication.

The set of Exchange Primitives contains not only the above set of Service Primitives but also the following Exchange Primitives which are derived from the [AES] definition of RPC Packet Types:

> **Ack** – indicates a client has received a response to an at–most–once request, and the server has been told to cease sending the response and discard the RESPONSE PDU.
>
> **Association** – contains either: a BIND and BIND_ACK, BIND and BIND_NAK, or ALTER_CTXT and ALTER_CTXT_RESP PDUs. It is associated with the actual setting up of a connection in a CO RPC transaction.
>
> **Association*** – contains either: a BIND and BIND_ACK, BIND and BIND_NAK, or ALTER_CTXT and ALTER_CTXT_RESP. It is associated with the second or more INVOKE and RESPONSE PDUs for the RPC transaction.

Exception – used in Connection–oriented RPC to indicate that the transaction contains either a FAULT, CANCEL, or ORPHAN PDUs.

Ping – used to indicate that a server has sent a WORKING PDU in reply to a PING PDU, and that the server is processing the client's call.

Response – basically the same as the Result service primitive.

Shutdown – used to indicate a termination of the connection, and the freeing of related resources for the RPC.

Unassigned_Fack – used to indicate that a packet was captured, whose source can not be identified as either a client or server. This would indicate a FACK packet captured before any packet with the client server information in it.

9.3 Connectionless RPC

Each RPC PDU consists of several fields which are given in [AES]. For CL RPC, we are interested only in two of these fields to identify PDUs belonging to a particular RPC. These fields are *Activity UUID* and the *Sequence Number.* The activity identifier is a UUID that uniquely identifies the client activity that is making a remote procedure call. The server can use the activity UUID as a communications key between it and the client. The sequence number is a 32–bit nonnegative integer which identifies the remote procedure call that an activity is making. Each remote procedure call invoked by an activity has a unique sequence number that is assigned when the call is initiated. All RPC connectionless PDUs sent on behalf of that particular call have the same sequence number, whether the PDUs are from the client to server or from the server to client. When an activity initiates a new remote procedure call, it increases the sequence number, so that each subsequent call has a larger sequence number. Together, the activity UUID and the sequence number uniquely identify a remote procedure call. Figure 6 shows a typical collection of packets and transaction hierarchy for a connectionless RPC protocol.

The following is the list of Exchange Primitives that are tracked by the reconstructor for the connectionless protocol:

1. INVOKE
2. RESPONSE
3. REJECT
4. CANCEL
5. ERROR
6. PING

7. NOCALL
8. ACK

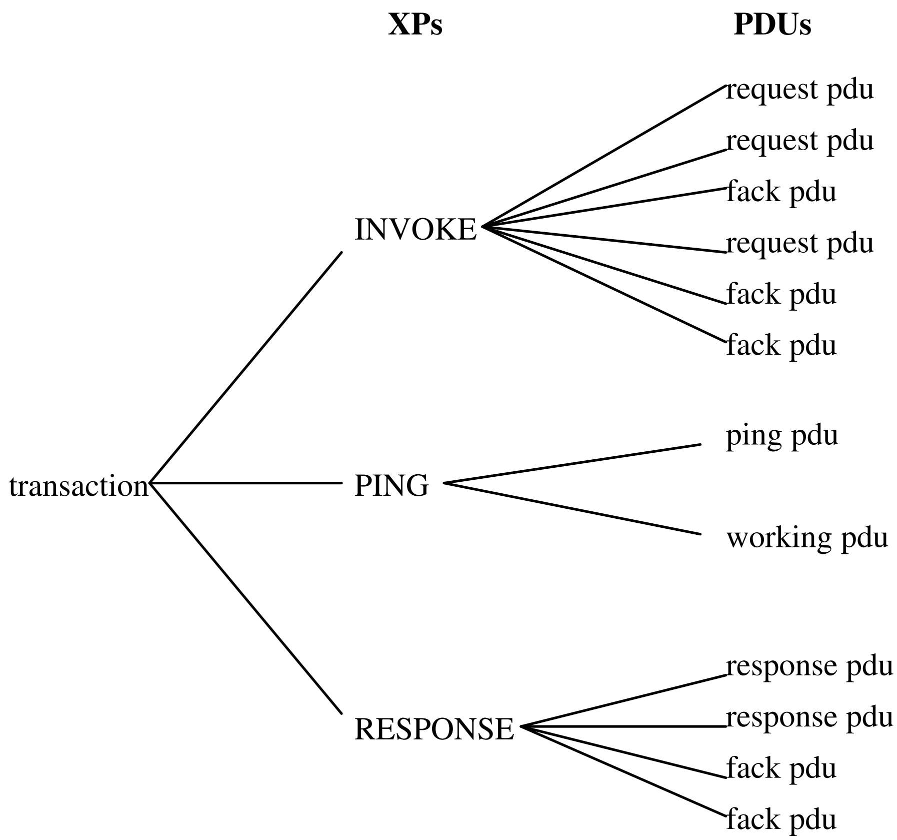

Figure 6. Example of a Transaction Collection hierarchy.

In a connectionless RPC there are ten different PDU types. These PDU types can be determined from the Packet Type field in the RPC header. These ten PDU types are as follows [AES]:

ACK PDU – A client sends an ACK PDU after it has received a response to an at–most–once request. An ACK PDU explicitly acknowledges that the client has received the response; it tells the server to cease sending the response and discard the RESPONSE PDU.

CANCEL_ACK PDU – The server sends a CANCEL_ACK PDU after it has received a CANCEL PDU. A CANCEL_ACK PDU acknowledges that the server has canceled or orphaned a remote call or indicates that the server is not accepting cancels.

CANCEL PDU – A client sends a CANCEL PDU when it has incurred a cancel fault. A CANCEL PDU tells the server to cancel or orphan a remote operation.

FACK PDU – Both clients and servers send FACK PDUs. A client sends a FACK PDU after it has received a fragment of a multi–PDU response. A FACK PDU explicitly acknowledges that the client has received the fragment. It may tell the sender to stop sending for a while. A server sends a FACK PDU after it has received a fragment of a multi– PDU request. A FACK PDU explicitly acknowledges that the server has received the fragment.

FAULT PDU – A server sends a fault PDU if an operation incurs a fault while executing on the server side.

NOCALL PDU – A server sends a NOCALL PDU in reply to a PING PDU. This reply indicates that the server is not currently processing the client's call. The server may have never received the request, or some of the request fragments may have been lost and need to be transmitted.

PING PDU – A client sends a PING PDU when it wants to inquire about an outstanding request.

REQUEST PDU – A client sends a REQUEST PDU when it wants to execute a remote operation. In a multi–PDU request, the request consists of a series of REQUEST PDUs with the same sequence number and monotonically increasing fragment numbers. The body of the REQUEST PDU contains data that represent input parameters for the operation. A REQUEST PDU may have one of the following types: idempotent, broadcast, maybe, broadcast/maybe, at–most–once.

RESPONSE PDU – A server sends a RESPONSE PDU if an operation invoked by an idempotent, broadcast, or at–most–once request executes successfully. Servers do not send responses for maybe or broadcast/maybe requests. A multi–PDU response consists of a series of RESPONSE PDUs with the same sequence number and monotonically increasing fragment numbers.

WORKING PDU – A server sends a WORKING PDU in reply to a PING PDU. This reply indicates that the server is processing the client's call.

The following pseudocode shows the logic used to build the Exchange Primitive for a connectionless RPC transaction:

```
int Get_Exchange_Primitive_CL ( int Packet_Type)

    switch (Packet_Type) {

    case RPC_PDU_REQUEST :
            ESTABLISH Activity UUID and Sequence Number
            return  INVOKE
```

```
case RPC_PDU_ACK:
        ESTABLISH Activity UUID and Sequence Number
        return ACK
case RPC_PDU_RESPONSE:
        ESTABLISH Activity UUID and Sequence Number
        return RESPONSE
case RPC_PDU_CANCEL_ACK :
        printf(”Cancel ACK Occurred \n”)
case RPC_PDU_CANCEL:
         ESTABLISH Activity UUID and Sequence Number
         return  CANCEL
case RPC_PDU_PING:
case RPC_PDU_WORKING:
case RPC_PDU_NOCALL:
        ESTABLISH Activity UUID and Sequence Number
          return  PING
case RPC_PDU_FAULT:
        ESTABLISH Activity UUID and Sequence Number
        return  FAULT
case RPC_PDU_REJECT :
         ESTABLISH Activity UUID and Sequence Number
         return REJECT
case RPC_PDU_FACK:
         if (Activity UUID and Sequence Number established)
              if (IP Source == Client IP Address)
                 return INVOKE
              else
                 return RESPONSE
          else
              return UNASSIGN_FACK
 default:
         printf(”packet UNKNOWN!! %d\n”,Packet_Type)
         return –1
}
```

9.4 Connection–oriented RPC

For connection–oriented RPCs, the PDUs do not have an Activity UUIDs to uniquely identify a remote procedure call. Therefore, a combination of source and destination hosts, source and destination port numbers and type of PDUs are used to uniquely identify an RPC. Figure 7 shows a typical collection of packets and the transaction hierarchy in connection–oriented RPC protocol.

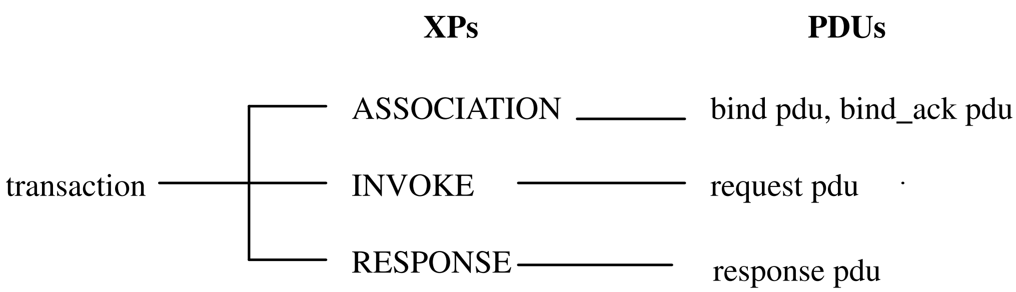

Figure 7. Example of a Transaction Collection hierarchy.

The following is the list of Exchange Primitives that are tracked by the reconstructor for the connection–oriented protocol.

1. INVOKE
2. RESPONSE
3. ASSOCIATION
4. ASSOCIATION*
5. SHUTDOWN
6. EXCEPTION

In connection–oriented RPCs there are eleven different PDU types. These PDU types can be determined from the Packet Type field in the RPC header [AES]:

ALTER_CONTEXT PDU – used to request additional presentation negotiation for another interface and/or version. The format is identical to the BIND PDU except that the value of the PTYPE field is set to ALTER_CONTEXT.

ALTER_CONTEXT_RESPONSE PDU – used to indicate the server's response to an ALTER_CONTEXT request. The PDU format is identical to the BIND_ACK, except that the value of the PTYPE field is set to ALTER_CONTEXT_RESPONSE.

BIND PDU – used to initiate the presentation negotiation for the body data, and optionally authentication. The presentation negotiation follows the model of the OSI presentation layer.

BIND_ACK PDU – is returned by the server when it accepts a bind request initiated by the client's BIND PDU.

BIND_NAK PDU – is returned by the server when it rejects an association request initiated by the client's BIND PDU.

CANCEL PDU – used to forward a cancel.

FAULT PDU – used to indicate either an RPC runtime, RPC stub, or RPC specific exception to the client.

ORPHANED PDU – used by a client to notify a server that it is aborting a request in progress that has not been entirely transmitted yet, or that it is aborting a (possibly lengthy) response in progress.

REQUEST PDU – used for an initial call request. The P_CONT_ID field holds a presentation context identifier that identifies the data representation. The OPNUM field identifies the operation being invoked within the interface. This PDU may also contain an object UUID.

RESPONSE PDU – used to respond to an active call. The P_CONT_ID field holds a context identifier that identifies that data representation.

SHUTDOWN PDU – is sent by the server to request that a client terminate the connection, freeing the related services.

The following pseudocode illustrates the logic used to build the Exchange Primitives for a connection–oriented RPC transaction:

```
int Get_Exchange_Primitive_CO (int Packet_Type)
    switch (Packet_Type) {
    case RPC_PDU_REQUEST
        ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
         return INVOKE
    case RPC_PDU_RESPONSE
        ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
         return RESPONSE
    case RPC_PDU_BIND
        ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
         return ASSOCIATION
    case RPC_PDU_BIND_NAK
        ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
         return ASSOCIATION
```

```
case RPC_PDU_ALT_CTXT
    ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
     return ASSOCIATION

case RPC_PDU_ALT_CTXT_RESP
    ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
     return ASSOCIATION

case RPC_PDU_SHUTDOWN
    ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
     return SHUTDOWN

case RPC_PDU_FAULT

case RPC_PDU_CO_CANCEL

case RPC_PDU_ORPHANED
    ESTABLISH  TCP Source/Destination Port & IP Source/Destination Address
     return EXCEPTION

default:
     printf ("packet Unknown!);
     return –1;
}
```

9.5 PP Design and Implementation

The logical flow diagrams for the PP CSC are shown in Figures 8, 9, 10 and 11. Figure 8 shows the logic for the main loop of the Packet Analysis Process. Figure 9 shows the logic used to read the Log file and create the output files (.tr, .xp, .pkt, .xpkt) used by the GUI. Figures 10 and 11 show the logic used for the analysis of CO and CL RPCs.

The output files contain transactions (.tr), exchange primitives (.xp), a list of packets (.pkt), and the hexadecimal dump of each packet (.xpkt). Together these files comprise a hierarchical indexing scheme used to traverse from the transaction level on down to the packet level. For example, the '.tr' file will indicate that for a given transaction, the corresponding exchange primitives are on lines 10 through 12 in the '.xp' file. Then, line 10, for example, can be used to index into the '.pkt' file to find a list of the packets used to build the XP. Subsequently, each packet listed in the '.pkt' has an index into the '.xpkt' file so that the actual data in a packet can be accessed.

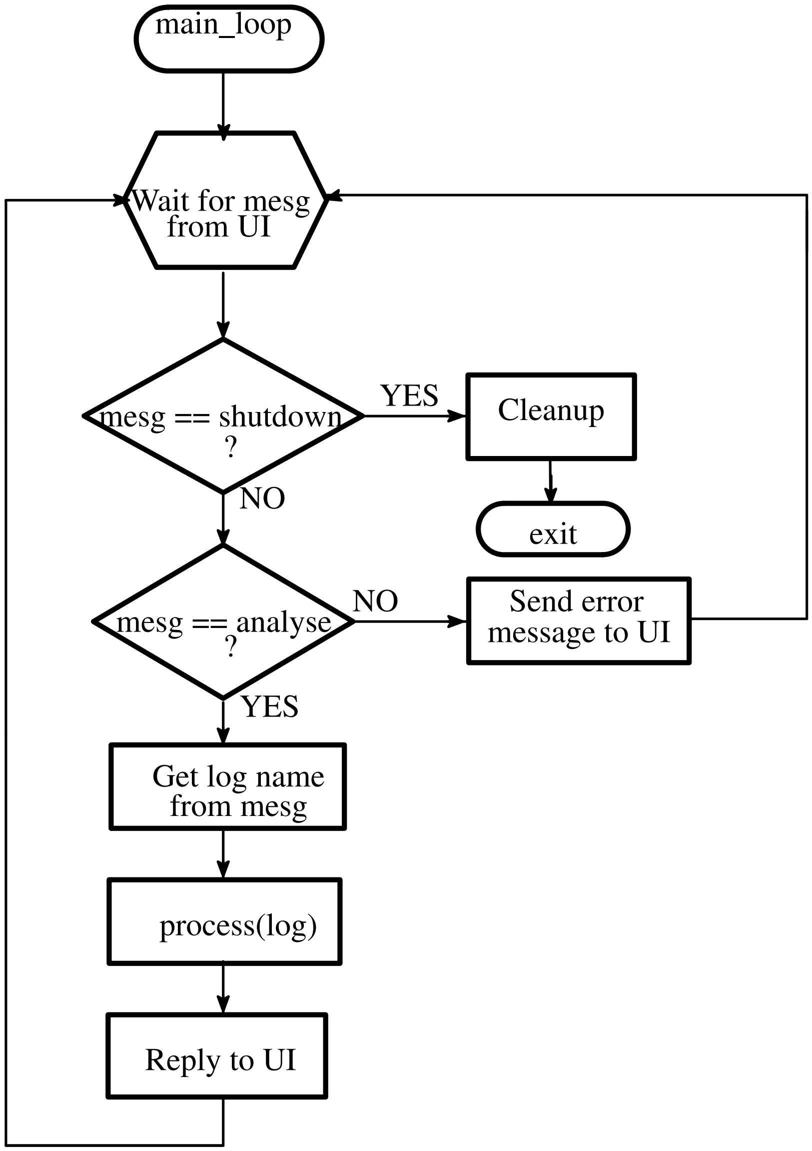

Figure 8. Main loop logic of packet collection.

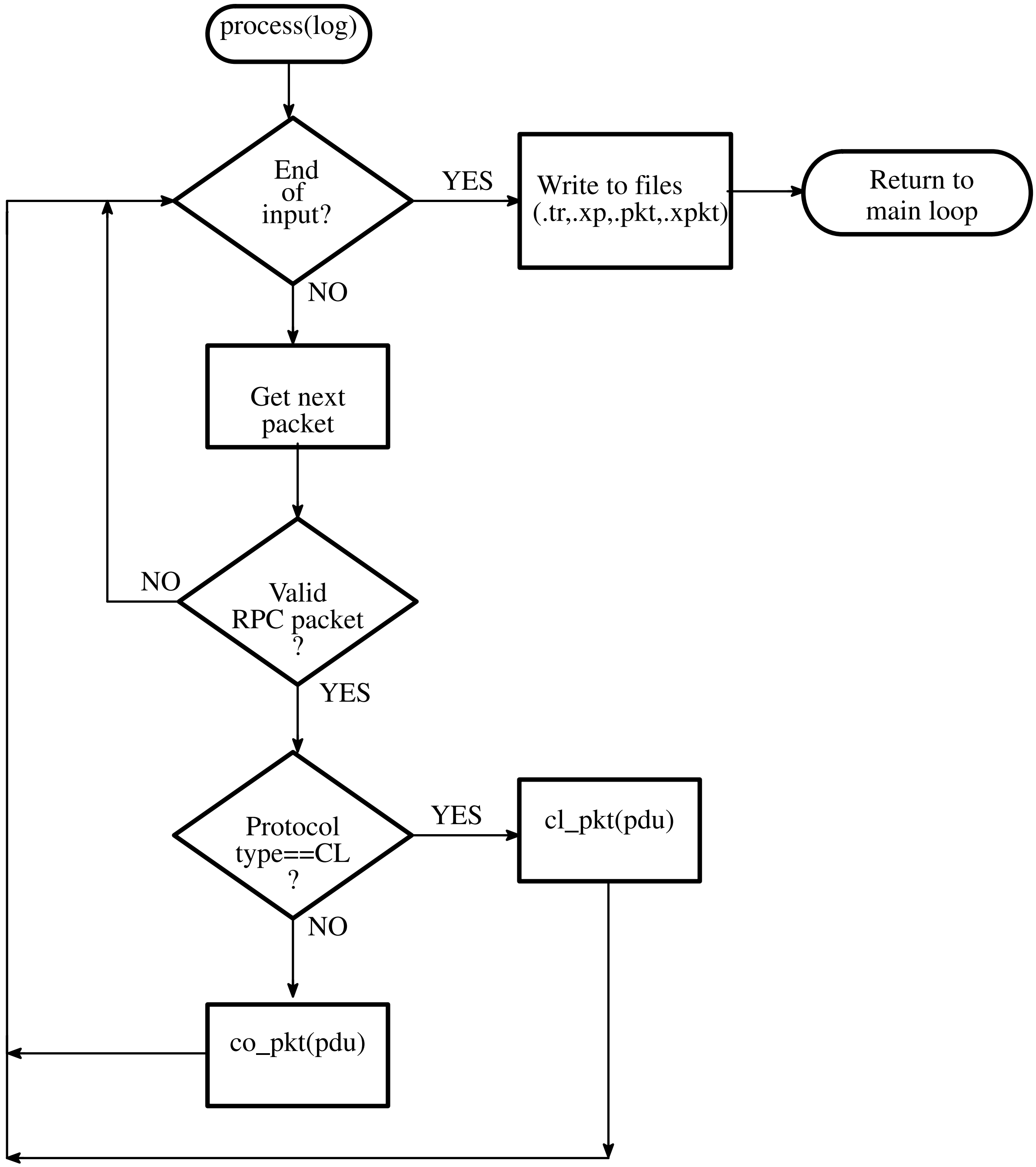

Figure 9. Logic for packet collection.

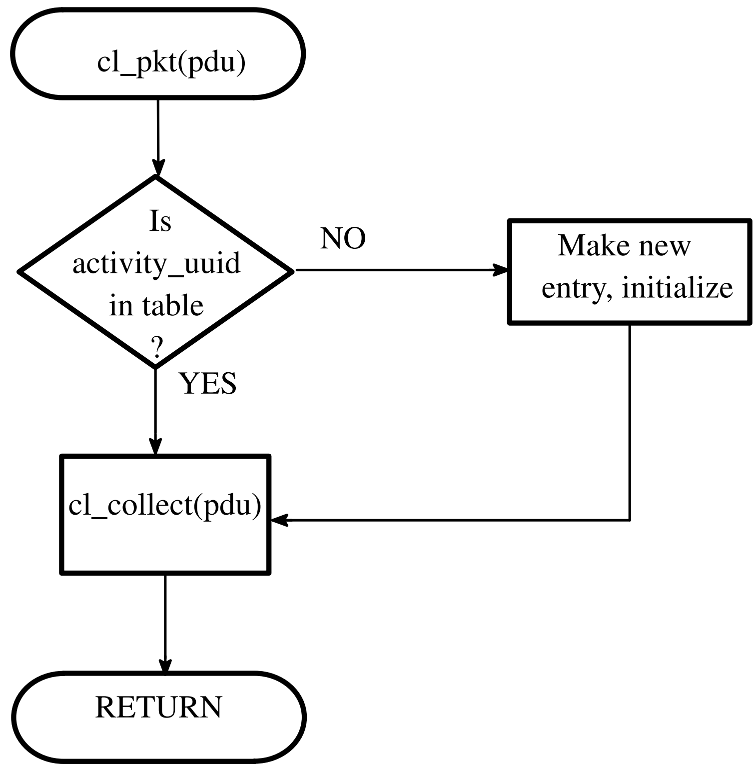

Figure 10. Logic for collection of connectionless packets.

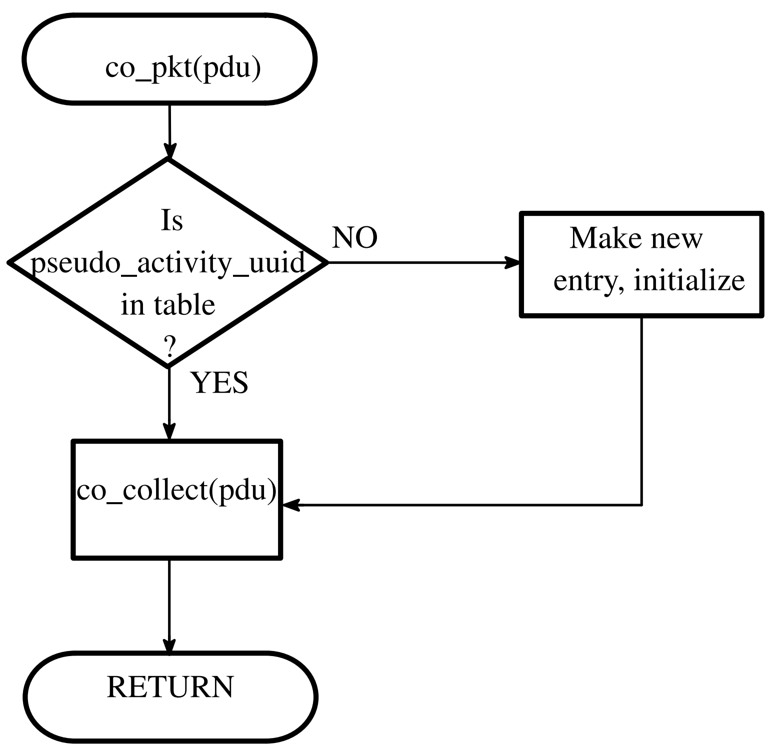

Figure 11. Logic for collection of connection–oriented packets.

The design of a three–level tree data–structure is shown in Figure 12. Figure 13 shows the structure of the packet list. This structure forms the basic building block for the RPC analysis. It is designed to capture the hierarchy of relationships between transactions, exchange primitives, and RPC PDUs.

For connectionless RPCs, there is an instance of this structure for every *activity* . An activity is an instance of a client application. It is uniquely identified by its Activity Universally Unique IDentifier (UUID).

For connection–oriented RPCs, there is an instance of this structure for every remote procedure call. It is uniquely identified by the TCP source port, TCP destination port, IP Source address, IP destination address, and a unique sequence number which is assigned by the Analysis CSU for each RPC transaction in the Log file.

A table consisting of these trees is maintained for every RPC:

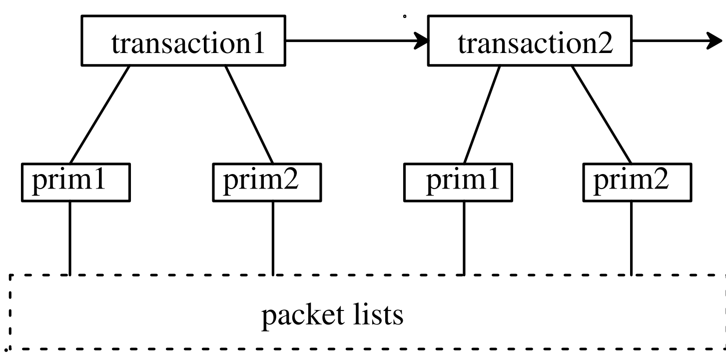

Figure 12. Three–level tree data structure.

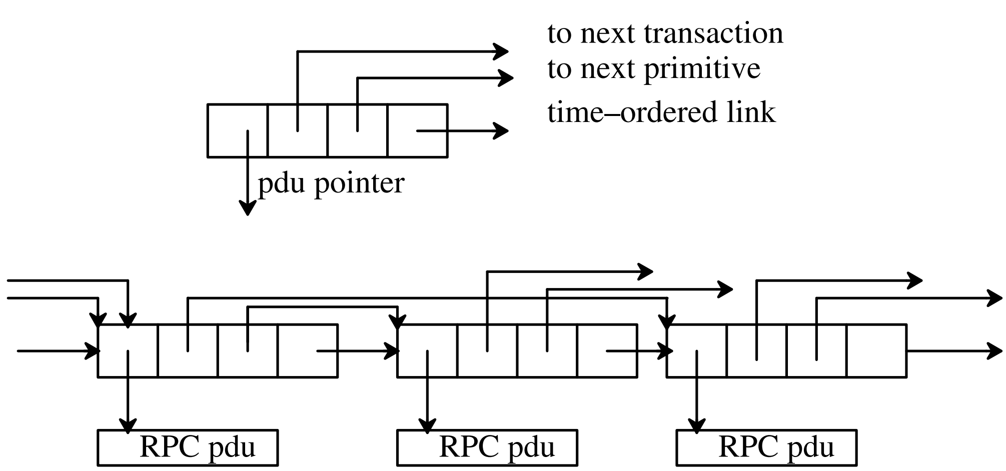

Figure 13. Structure of packet lists.

Each level of the tree consists of linked lists. The need for linked lists arises because there is no way to predict the nature of the traffic beforehand. All the linked lists have a head and tail pointer. The use of tail pointer helps in the quick addition of new members into the list.

The transaction list forms the first level in the tree, and it has pointers to a set of primitives. Presence of a primitive in a transaction indicates that the RPC consisted of some RPC packets that caused the primitive to occur. The primitives themselves lead to the actual RPC packets seen on the network segment. Each *packet_t* structure has provision for three pointers, forming three logical linked lists. The first linked list is time–ordered. The packets are chained in the order they are seen on the network segment. The second linked list links all the packets for a transaction in transaction–order, and the third links all packets belonging to a primitive. In addition to keeping track of the packets and queueing them appropriately, the Analysis module also establishes the client and server host names.

The salient feature of this data structure is that it allows for a rich suite of operations to be performed. The operations are listing of packets in time order, listing the packets for a given transaction, listing of the primitives that have occurred for a given transaction, listing of just the primitives for a given set of transactions, and so forth. At this time, listing of packets is done in two formats. The first format is called **Transaction Time Order (TTO)**, in which the listing of packets depends on the transaction's sequence number. In this format, a transaction, its corresponding XPs, and packets corresponding to the XPs are listed in order. The second format is called **Packet Time Order (PTO)**, in which the listing of packets depends on the time at which the packets are captured. In this format, first a packet, then the XP to which it belongs, and then the transaction to which that XP belongs are listed in order. For TTO, two different types of files are generated, depending on the user requirement (either for display or print). In the current version, packet time order files are generated only for printing.

A table is maintained called *act_tab* to put all the transaction lists together. The *act_tab* keeps track of all the transactions for CL, as well as CO.

The next version of SMT will support multiple RMON devices on the network, so there exists the possibility of having multiple copies of the same RPC PDUs. Thus, each of the received packets is qualified by the RMON agent's identifier. A list of the packets belonging to an activity and a RMON identifier is called a *chain*. The Analysis module will keep track of each chain received from the RPCAP CSC. This is done by maintaining separate list–trees for each stream, for every activity.

10 GUI CSC

The GUI CSC is built using the X Window System, Release 11 Version 5, and uses user interface elements called *widgets* supplied with the OSF/Motif toolkit. The user interface is written to conform to the OSF Application Environment Specification – User Interface document. The GUI process reads the configuration file and starts the RPCAP and PP processes during initialization. The user interface and logical flow diagram of GUI CSC are shown in Figures 14 and 15, respectively.

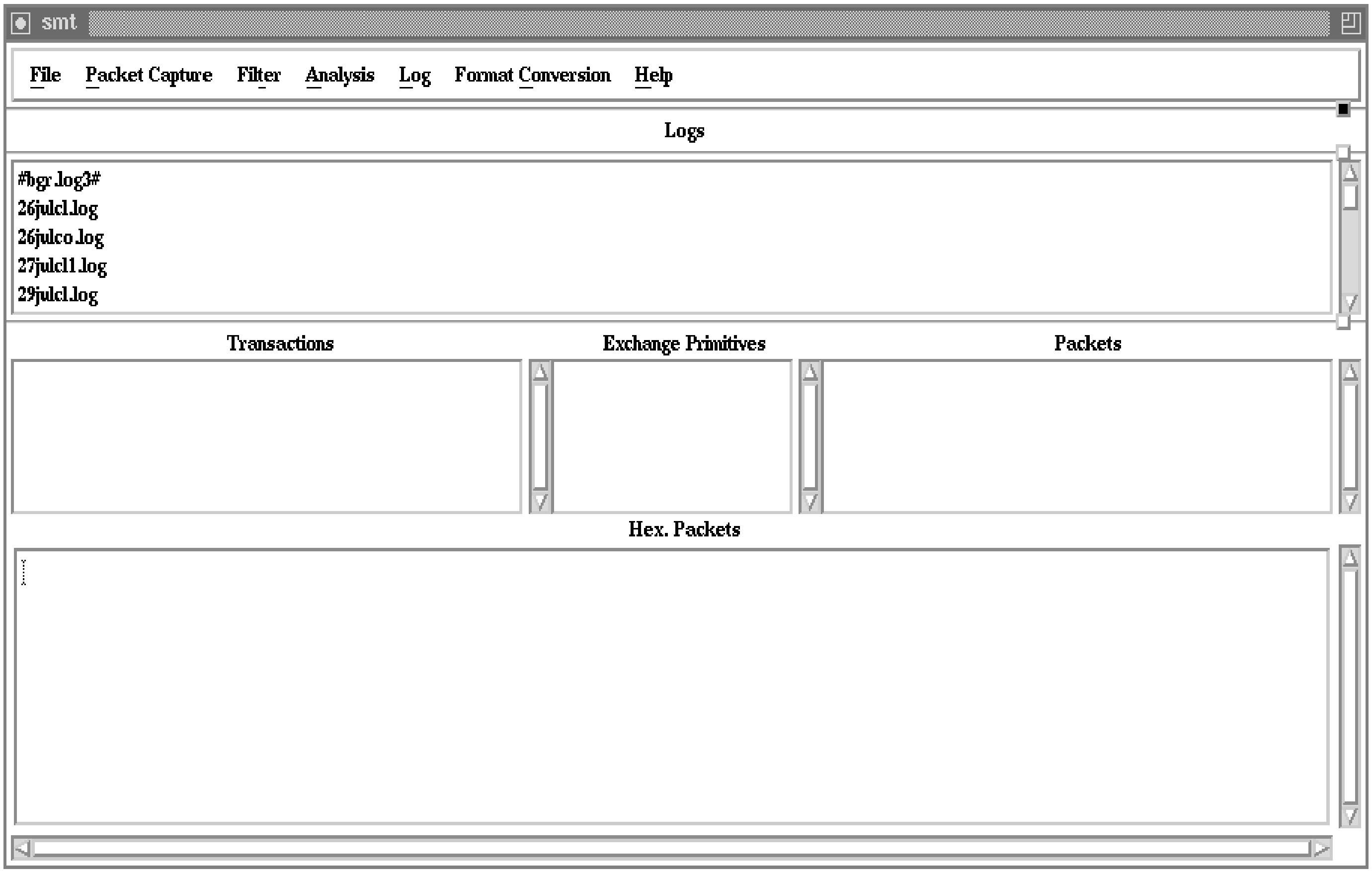

Figure 14. User Interface for SMT.

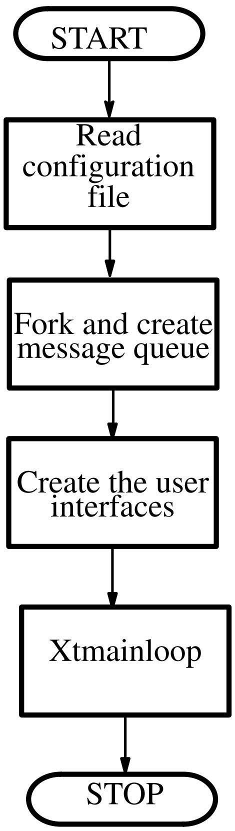

Figure 15. Flowchart for main program.

The GUI CSC also creates the message queue necessary for the communication between the other CSCs. The GUI CSC is comprised of 5 CSUs, as shown in Table 3.

Table 3. CSC GUI – Summary of CSU Components

Name	Description	ID
CSU RPCAP Access	User interface functions for configuration and control of RPCAP CSC	CSU 3.1
CSU Analysis Access	Implements functions useful for transaction tracing and decoding of packets.	CSU 3.2
CSU User Interface	Creates the user interface and sets resources for the various display elements.	CSU 3.3
CSU Format Conversion	User interface functions for conversion of other formats into the format understood by SMT PP.	CSU 3.4

CSU Post Analysis Filter Access	User interface functions to get filter information, call filter, and display results.	CSU 3.5
CSU Help	Provides on–line help on how to use the DCE SMT system.	CSU 3.6

The GUI CSC consists of several CSUs, which are basically interfaces to the different CSCs. The following sections explain the function of each CSU.

10.1 RPCAP Access (CSU 3.1)

The RPCAP Access allows the user to configure RMON devices to capture RPC traffic on its subnet. This allows packet capture in the following modes.

1. Capture all CL RPC packets on the network.
2. Capture CL RPC packets between two hosts.
3. Capture CL packets to/from a particular host.
4. Capture all CO RPC packets on the network.
5. Capture CO RPC packets between two hosts.
6. Capture CO packets to/from a particular host.
7. Capture all RPC packets on the network.

Also, the configuration involves the following information:

1. RMON IP address.
2. Log File name.
3. Host 1 IP address.
4. Host 2 IP address.
5. Capture mode.
6. Flag indicating whether capturing is occurring or not.

This CSU allows the user to send *start* and *stop* commands to the RPCAP CSC.

10.2 Analysis Access (CSU 3.2)

The Analysis Access implements functions useful for transaction analysis, decoding of packets, and so forth. The information obtained from the analysis access is helpful in studying the client–server interaction. The user can analyze the RPC traffic in three levels. They are as follows:

1. Transaction level,

2. Exchange primitive level,
3. Packet level.

The User Interface invokes the PP module when the Display option is chosen under Analysis. A message is sent to the PP to analyze the log file currently selected in the list of log files. The User Interface waits for the return message.

If the returned message indicates a successful analysis, the User Interface then accesses the four output files from PP. These files, as discussed in Section 9.5, contain information regarding the transactions observed in the log file.

The User Interface gains access to the information in these files by essentially copying the files into memory. Each file is read into an array. The array sizes are allocated, depending on the number of transactions, exchange primitives, packets, and individual packet sizes.

From these arrays, the User Interface can recreate the hierarchy of any transaction chosen by the user from the transaction level, all the way down to the display of the hexadecimal data of a packet in that transaction.

Another option under Analysis is Edit UUID Mapping. This feature allows the user to substitute names for hexadecimal UUIDs, which are 32 digits. By doing so, the translation fields in the individual packet window output is made more readable. For instance, instead of an Object UUID reading `0x0a1a3b2a39867123a67238c8b9b2312a`, it could read `UUID_one`.

This mapping from UUIDs to names is maintained in a linked list during program execution, but remains in the configuration file *.smtrc* when SMT is not running. The UUID mapping is read in from the configuration file when SMT is invoked, and is written every time Edit UUID Mapping is used.

Edit UUID Mapping allows the user to add UUID to name mappings, delete them, or change them.

10.3 Graphical User Interface (CSU 3.4)

The main user interface, as well as the other interfaces are created by this CSU. The main user interface – which consists of a menubar, log window, Transaction window, Exchange Primitive window, Packet window and Hexadecimal Packet window – are created and managed in this CSU. The user interface elements are created by making calls to Xm and Xt libraries. The access to other CSU's in the CSC GUI is done in this CSU. Resources are set for the size of the interfaces. Also, callbacks are associated with appropriate menu items. These are functions which are to be invoked when a given action is performed with the mouse on a particular widget, such as pressing a button.

10.4 Post–Analysis Filter

This CSU contains user interface functions to get filter configuration information, invoke the filter, and display the results. There is a desire to implement several types of filters eventually, but there is currently only one filter, a Service filter.

The filtering is done using an expression comprised of a list of tuples OR'd together. Each tuple consists of several fields. The fields inside a tuple are logically AND'ed. The User Interface provides the functions necessary to get this information from the user in order to build the expression. Each tuple corresponds to one popup configuration window. In the popup window, all of the possible fields are listed. The user fills in specific fields they want to filter on. The fields are listed below:

1. Object UUID.
2. Interface UUID and Interface Operation.
3. Binding protocol sequence, server IP address and port, and client IP address and port.
4. A *Do Mapper?* flag. When "YES", and a port is specified, it indicates to filter all packets with the same information, but for port 135.

The user can build an expression by specifying multiple tuples. When the filter is invoked, the PP is first called to build the array structures in memory for the transactions in the current log file. Then, the list of tuples is passed to the filtering module. The filtering module then builds the expression from the linked list, and filters on a transaction level of granularity. When a transaction passes through the filter, that transaction is marked "YES" by the filter.

There are two, user–configurable modes in which the filter results can be displayed. In **partial** mode, only the transactions marked "YES" will be displayed. This mode allows for nested filtering because any subsequent filtering is done on the transactions displayed from the initial filter, not on the original list. In **marked** mode, all of the transactions are displayed, but the ones that passed the filter are marked with an asterisk (*). Subsequent filtering is done on the entire list, so this mode does not support nested filtering.

10.5 Format Conversion(CSU 3.5)

This enables the user to convert log files which are in other formats into the format understood by PP of SMT. The two formats currently supported are, Etherfind and Iptrace. The user enters the Input filename to be converted, and the filename of the output SMT format file.

10.6 Help (CSU 3.6)

This provides on–line help on how to use the system. Help is available on all modules. The help module details the following:

1. How to accomplish a particular task.
2. The set of valid inputs.
3. What the user should do if some error occurs.

Help is available from the main menu on all the user commands. Context–sensitive help is also available.

11 COMM CSC Design

COMM CSC is defined as the inter–process communications component of the SMT. CSC COMM does not exist as a separate program but as a common message queue interface and message definitions for all the SMT CSCs.

The CSC COMM component is written as a separate CSU which is compiled and linked into each SMT program (PP, RPCAP, GUI). Common appropriate header files are provided for the source code to ensure common access to communications functions.

CSC COMM is implemented as a POSIX–compliant, kernel–based message queue service with specifically defined message types for each combination of CSC source and CSC target. A common message queue may be used for both reading and writing by all CSC processes. CSC COMM defines not only message types, but also, standard message formats and data structures which are agreed upon by the CSC components.

As an additional note, a recent release of the first multiple–RMON SMT (SMT.99mt) included a new scheme for communications. The message queue in SMT.99mt was eliminated because the entire SMT program is contained in one process. The PP is implemented as a function call from the GUI. Packet capturing is done with multiple RMONs by using threads, one for each RMON.

12 Current Status

The current version satisfies all of the basic design goals for SMT. It runs on IBM/AIX and Sparc SunOS 4.1.3. It is able to communicate with different types of RMON agents on different MAC layers (Ethernet IEEE 802.3 and Token Ring IEEE 802.5) and on different network segments. It captures and analyzes both connection–oriented and connectionless RPC PDUs, and displays them in Transaction Time Order and Packet Time Order formats.

Another capability is the *filter*. This feature allows the user to display only selected transactions. The filtering can be done on object UUIDs, interface UUIDs or operation numbers, and client/server addresses/ports. Also, the *Do Mapper?* flag can be configured "YES" when a port is specified to filter on packets using the same tuple information, but with a port of 135, instead of the specified port.

Also, there are three utilities available in the current version of SMT – *format conversion*, for converting log files in SMT–readable format, *uuid2cfg*, for automatically scanning a specified DCE.IDL file for a UUID name mapping that can be added to the configuration file, and *Edit UUID Map*, which allows the user to update the UUID mapping stored in the configuration file.

As discussed in Section 11, the most recent release of SMT, known as SMT.99mt, is the first SMT release to have multiple RMON capability. This is achieved by using threads, one for each RMON. Also, SMT.99mt implements PP as a function call from the GUI. Because of this, and *threads* that are used for RPCAP, there is no longer any need for the message queue form of communication. All components of SMT.99mt are contained within one process, so communication can be done via global variables.

13 Future Work

The recent release of SMT.99mt essentially signalled the beginning of the Phase 2 development of the SMT project. The goal of Phase 2 is to implement an SMT that has the capability of threaded, multiple–RMON packet capture, and threaded packet analysis. The threaded multiple–RMON capability is present in SMT.99mt, but threaded packet analysis is not. Also, a log file browser will be implemented to allow a user to analyze multiple log files at once. In addition, a statistics feature will be added to allow a user to track the types of packets being captured, and so forth.

Phase 3 of the SMT project will involve adding real–time analysis of transactions, and real–time filtering to SMT.

14 Conclusions

SMT was developed to help DCE administrators and programmers capture packets and analyze them at various semantic levels. SMT reduces the workload required to debug packets. It can easily be modified for application in other distributed environments. And also, the modular design of SMT helps in using each of its components independently for different applications.

References

[AES] *AES/Distributed Computing–Remote Procedure Call*, Revision A–Review Draft, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142.

[HOW93] Howe, James, *An Environment for 'Sniffing' DCE–RPC Traffic*, CITI Tech Report 93–4, Center for Information Technology Integration, The University of Michigan, June 1993.

[LYN93] Lynch, D. C., and Rose, Marshall T., *Internet System Handbook*, Prentice Hall, 1993.

[ROE91] Roetzheim, William R., *Developing Software to Government Standards*, Prentice Hall, Chapter 9, 1991.

[Stallings] Stallings William , *SNMP, SNMPv2 and CMIP*, Addison–Wesley Publishing Company Inc., 1993.

[BTNG] Jan van Oorschot, Data Network Performance Analysis Project, CARDIT, Delft University of Technology, Netherlands. e–mail: JPMvOorschot@et.tudelft.nl,