Efficient Communication Strategy of Enterprise TCL/TK Application with Multi Process System:-A Study

Kumar Gaurav, Tushar Gupta, Madhur Bhatia

Mentor Graphics Corporation

Abstract

The GUI tool of Veloce emulation system is a TCL/TK based application. The Veloce software has a complex multiple process distributed architecture. The Inter-Process-Communication (IPC) within the software components involves frequent and bulky data transfers between the processes. VeloceGUI on one hand needs to update its state very frequently based on responses from some of the software components and emulation runtime system, and on the other hand needs huge on demand data transfer from other set of servers.

The paper elaborates how to use different communication methods to get maximum performance with minimum memory utilization. The paper also discusses how the TCL/TK based GUI interacts with larger client-server ecosystem, communicating with each other, using a sophisticated message passing system

Glossary

GUI – Graphical User Interface

IPC – Inter Process Communication

RDS – RTL Data Server

WDS – Wave Data Server

1. Introduction

There are two types of communication mechanism used by VeloceGUI: –

i) TCL Sockets
ii) Message passing library built over C-Sockets

Socket Communication consists of two steps:

i) Exchanging of data
ii) Processing of data

Exchanging and processing of data between client and server through socket communication, involves lot of challenges such as optimization of time, speed, memory and maintaining backward compatibility. Processing of data requires, parsing of data to find the actual command, that client has passed to the server for processing. Parsing of data may take significant amount of time if the frequency of communication is high; however this can be optimized to get fast response from the server

VeloceGUI communicates with RTL-Data Server (RDS) and Wave Data Server (WDS) processes through raw socket interface using the TCL library functions, as these involve bulk data transfers, of rtl design connectivity information and waveform data. This interface is optimized for data transfer efficiency, as the volume of data is huge. The frequency of communication is generally on demand and numbers of

commands are lesser between GUI application and RDS/WDS. Thus the time overhead of command parsing is minimal and therefore bulk data is transferred in most efficient way. At the same time most of the intensive databases loading and data processing tasks are out-sourced to the servers, reducing the overall memory foot-print and response time for VeloceGUI application.

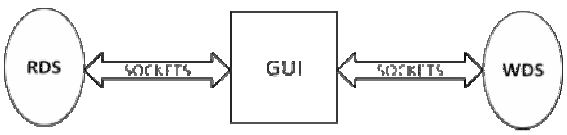
For interactive emulation control and communication, VeloceGUI uses C/C++ API interface (also called VeloceAPI), which is a shared library system. The VeloceAPI system interacts with other components of Veloce runtime system, using a message passing system (called messaging systems) built over C-sockets. This communication interface is mainly designed for fast interactive response, as the number of command and communication frequency is larger between GUI and Veloce runtime system. The design eliminates time spent in parsing the command level data through interface definition mechanism. The messaging system also takes care of maintaining backward compatibility within different servers/clients. The Messaging System sockets are registered in the TCL Event Loop to enable continuous polling on the messaging system sockets without blocking the GUI.

In this paper, we will discuss the various aspects of communication of GUI with RDS/WDS using TCL raw sockets and Veloce runtime system using message passing system built over C-sockets. We will also discuss the issues that were resolved during the development.

2. Communication between GUI and RDS/WDS over raw sockets

RDS and WDS are rtl-data-server and wave-data-server. These servers are meant for storing large databases corresponding to RTL design hierarchy and their waveform data.

Our emulation GUI shows the RTL design hierarchy and waveform data in hierarchy/signal browser and wave browser respectively. For the population of hierarchy/signal tree and waveform data, Veloce GUI communicates with RDS and WDS through raw sockets. GUI initiates the socket connection between itself and RDS/WDS process when the need arises i.e. when the first query arises for the RDS/WDS. Until then there is no connection between these processes. After the connection is set, GUI creates different commands and send them to RDS/WDS. These commands are used to populate the RTL design tree structure for hierarchy/signal browser and waveform data for wave browser. As the command reaches the RDS/WDS, it parses the command, process and fetches the corresponding data and provides the data to GUI. GUI remains in blocking state till the processing is completed by RDS/WDS. As the data reaches the GUI end, GUI populates its corresponding database and displays the results in hierarchy/signal browser if provided data is from RDS, or in wave browser if provided data is from WDS.



1.a *Communication of GUI with RDS/WDS using Sockets*

The code snippet shown below illustrates, how the connection establishes between GUI and RDS/WDS

```
# Adding signal to wave window.
$wave_data_obj add_to_wave $hierarchy
```

This proc will first check that the connection is established between wave data server and GUI or not. If not then it will connect both the servers and then will send the command.

```
itcl::body
wave_data_server::add_to_wave  {args} {
if {$d_wave_server_id} {
  if {[catch {eval $this wave_server \
    $args} msg]} {
       return “error $msg”
    }
  } else {
    if {[catch {wave_server_connect} \
      mesg]} {
        return $mesg
      }
      set d_wave_server_id $mesg
      if {[catch {eval $this \
        wave_server $args} msg]} {
           return “error $msg”
      }
  }
}
```

In the code above when the first query arises for the wave server i.e. add signal to wave, then before sending the command to the corresponding servers, GUI checks for the server id, if it exists then GUI sends the command otherwise it establishes the connection between itself and the server and then executes the command.

The communication between GUI and RDS/WDS is a blocking communication, it means that GUI will have to wait till RDS/WDS processes and provides the data. When a user submits a request, he has to wait for that request to complete. As RDS and WDS are dedicated database servers serving the GUI only, so providing the data to GUI does not take much time. When user submits the tasks, GUI creates its corresponding command and sends to these servers. These servers fetch the data and send the results back to GUI without taking much time.

These are the list of tasks, which need communication over raw sockets with RDS/WDS

- Expanding any hierarchy in design hierarchy tree.
- Searching all the signals of a module.
- View designs in schematic and netlist graphical view.
- View waveform of signals in wave window.

When user gives any such task to GUI, then user does not want to wait for the notification from the GUI- about finishing of the tasks, but user wants to see the results immediately and will not mind if he is blocked from submitting new requests for the small time interval during which the request will be served.

The frequency of communication and number of commands are lesser between GUI application and RDS/WDS. These servers are created only to entertain the user tasks, for which user want to see the result immediately. The syntax of these commands is also simple, thus the time overhead of command parsing is minimal, therefore bulk data is transferred in a most efficient way. RDS/WDS read big intensive databases so their loading time and data processing time remain outside GUI bring up time, reducing

the overall memory foot-print and response time of VeloceGUI application.

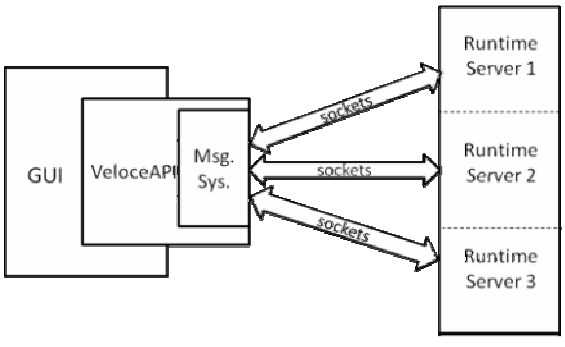
3. Communication between GUI and Veloce Runtime System using Message Passing System

Veloce Runtime System interacts with VeloceGUI using C/C++ API interface (also called VeloceAPI), which is a shared library system. VeloceAPI system interacts with other components of Veloce runtime system, using a message passing system built over C-sockets.

Emulation GUI, apart from displaying design hierarchy and waveform, also does many critical tasks, which are generally required from the GUI of Emulation product. These tasks involve

- Compiling the RTL design
- Keeping the GUI state updated
- Running the Emulation
- Downloading the Emulation database
- Downloading the Memory in the design
- Downloading and Updating the Trigger into hardware
- Getting and Setting the value of the register
- Adding break points in the design

For executing these tasks GUI communicates with Veloce runtime system through VeloceAPI, which is dynamically linked shared object library. GUI access Veloce runtime system through a message passing system (Messaging System) built over C sockets. The communication APIs are generated using a sophisticated compiler and socket management is done internally inside the messaging system library. The messaging system library provides the mechanisms to register the messaging system sockets to the event loop of GUI developed in TCL/TK. As these sockets get registered in TK event loop, then all the functions in Veloce runtime system can be accessed through VeloceAPI interface by the GUI and through the Messaging System interface by the VeloceAPI. All the calls to access the VeloceAPI functions are asynchronous calls. The VeloceAPI manages the launching/terminating of Veloce runtime system. The VeloceAPI uses a predefined protocol with the GUI to unblock while it is waiting for the data from the Veloce runtime system. Thus the GUI does not wait till the processing of the task is done by Veloce runtime system. GUI can be used for other purpose, till the time callback comes from Veloce runtime system. GUI remains in non blocking state.



2.a *Communication of GUI with Veloce Runtime System using Messaging System*

The code snippet shown below illustrates, how GUI communicates with Veloce runtime system using VeloceAPI message passing system

// VeloceAPI Interface Code

```
int    RTS_evalcmd    (ClientData    cld,
Tcl_Interp  *intrp,  int  argc,  char**
argv) {

    ……….
    ……….
    sts = RTS_eval (argv[1]);
    RTS_WaitForCallbak(wait);
    If(wait == true){
      Tcl_SetVar(intrp,hastowait,”1”,
        “TCL_GLOBAL_ONLY”);
    } else {
      Tcl_SetVar(intrp,hastowait,”0”,
         “TCL_GLOBAL_ONLY”);
    }
    ……….
    ……….
    Return sts;
}
```

TCL CODE

```
incr task_queue 1
set retval [RTS_evalcmd $cmd]
if{$retval !=0}{
    return –code error $retval
    incr task_queue -1
}
if {$retval == 0 && $hastowait == 1}
    wait_for_callback
    release_prompt
} else {
    incr task_queue -1
}
```

In the code above, the GUI process invoked the commands of VeloceAPI with the task that needs to be processed in Veloce runtime system. VeloceAPI interface just sends the task to Veloce runtime system. The Veloce runtime function accepts the task and sends the acknowledgement for the acceptance. A callback function is registered which is being called when the processing of task is being done by the Veloce runtime system. If the processing of the task requires time then TCL global variable “hastowait” is set by the VeloceAPI interface. After the status is being returned back to GUI then GUI just checks the return status and value of “hastowait” variable. If it is set then GUI called the proc wait_for_callback and releases the prompt. Now polling is started at the GUI sockets. The wait_for_callback proc does vwait on the variable which is being reset in the callback function. The prompt is released and can be used for other tasks.

The list of tasks which is allocated to the Veloce runtime servers are of the category for which user can wait for their completion. These tasks are run in the background without hindering user interaction with the GUI. User can use the GUI for other purpose like exploring the rtl hierarchy and looking at the waveforms, while these tasks are being processed in the background. The processing of these tasks should not block the GUI.

The processing of these tasks is in background but it does not mean that user can wait for long to see the results. User wants a quick response and notification from these servers. The number of commands and communication frequency are larger between GUI and Veloce runtime system thus for optimizing the time, our design eliminates the time spent in parsing the command level data through interface definition mechanism. This communication interface is mainly designed for fast interactive response.

4. Issues taken care during development

During the development of the GUI, we encountered number of issues. Two of the major issues were

i) Maintaining backward compatibility

ii) Optimizing the time and memory for blocking servers.

Maintaining backward compatibility is the major issue which needs to be handled carefully. Any change in the interface part of the GUI results in breaking the compatibility. There should always be synchronization between the servers and the GUI process. But if the forward and the backward compatibility are to be maintained between the servers and the GUI process, then instead of synchronizing the GUI and the servers, you also need to support the parsing of the older commands by the new server and parsing of the older data from the new GUI process. This is the important point that needs to be taken care of.

Communication between GUI and RDS/WDS server is blocking. The GUI will be in hung state till the RDS and WDS servers are processing the data. The busy state of the GUI increases the impatience in user. So it is the important task of the developer to optimize the processing and fetching time of the data. RDS and WDS are database servers. These servers just parse the command, fetch the corresponding data and send it back to GUI. As frequency of commands are lesser thus the time spend in the parsing is minimal. Developer should concentrate on minimizing the fetching time of the data. The fetching time can be minimized, if the data is stored in a proper container, which decreases the complexities during the search. Decreasing the complexities by using the proper container, increase the response time of the servers.

5. Conclusion

Performance of the GUI can be increased by selecting the right communication methods between the processes. If the communication between two processes is less and the syntax of commands are simple then GUI can communicate with the server using raw sockets. If the frequency of communication is large then communication between the processes can be done using message passing system.

Interface compatibility between the processes also needs to be taken care of during the development. Thus any change in the server side of the interface should be reflected in the client side as well and vice versa.