Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

A General Infrastructure for Communication between Petri Modules

An approach based on GPenSIM

Master's Thesis in Computer Science

By
MD Suhel Ahmed

Internal Supervisors
Dr. Reggie Davidrajuh

June 15, 2021

Abstract

Modularization provides many benefits to real-world discrete-event systems, such as flexibility, comprehensibility, and robustness. Davidrajuh[2] presents a new modular Petri net with well-defined Petri modules. With the new modular Petri net, distributed Petri modules are designed. The modules are hosted on different computers that are geographically kept apart but can communicate over the network. The distributed Petri modules approach minimizes the simulation time; reduces the state space size and other complexities. The distributed Petri modules are implemented in the software known as General-purpose Petri Net Simulator (GPenSIM).

In this thesis, a network layer is implemented into GPenSIM to allow the distributed modules to communicate. Two distributed systems are designed and implemented in GPenSIM, and MATLAB TCP/IP socket communication is used to develop the network. Implementing a network layer into GPenSIM allows simulation that is more realistic and easy to analyze the performance.

Acknowledgments

I am grateful to my supervisor, Dr. Reggie Davidrajuh, for giving me the chance to work on my thesis under his supervision. Despite his hectic schedule, his help, guidance, support, and helpful suggestions and advice were vital throughout the thesis.

Contents

Chapter 1

1 Introduction

In a distributed architecture, components are displayed on separate platforms, and vast number of components might collaborate through a communication network to achieve a specific purpose. The client-server architecture is an example of a distributed system. Distributed systems are built on the principles of transparency, dependability, and availability.

The project deals with discrete event dynamic systems (DEDS), which are asynchronous dynamic systems in which discrete events in the system trigger state transitions [1]. Manufacturing and communication systems are only two examples of dynamic systems that feature a DEDS framework. More applications may be added to the DEDS framework due to the state space method for expressing and evaluating such systems. In developing the state space method to studying DEDS, it will be assumed that some of the system's events are controllable, i.e., may be activated or deactivated. The purpose of DEDS control is to direct the system's behavior in a way that we find desirable. However, it's also believed that we can only view a subset of the events, i.e., we can only view part of the events that are taking place in the system, not all of them. Therefore, we may be obliged to make choices about the system's condition and regulate a DEDS based only on our observations in some instances.

There is booming progress in DEDS areas; as a result, it may genuinely describe and solve some issues in manufacturing systems, communication, and computer networks. However, many natural systems which need to be studied by DEDS are usually very complicated. The system describes at three different levels, logical level, timed models, and stochastic performance level from the different views, analyze its behavior and performance and solve the control problem [2]. An important area of the logical level DEDS research is the Petri Net formalism. The most remarkable topic is the development of high-level PN. The spirit is to specify more structured content for the tokens, thus reducing the number of places and the complexity of the PN operations.

Petri net is a handy tool for modeling discrete-event systems. Many application areas of Petri net, such as performance evaluation and communication protocol design, are successful applications. Up-and-coming applications model and analyze distributed software systems, distributed databases, flexible manufacturing, and industrial control systems [2]. However, Petri's new models of real-life systems are enormous, even for a simple system, and their state spaces are usually of infinite size. Due to the vast size of the model, analyzing the model for its structural and behavioral properties become difficult. The most important and useful property of Petri nets is their explicit state space. The state space is automatically generated, showing every possible state that can be eventually reached from an initial state. Model-checking from the huge or infinite state space is often difficult, if it is possible at all. During simulations, the tokens in a Petri net have to go through every transition and place on their path. In addition, the transitions have to be checked for their enabled and the other firing conditions from the environment that makes the simulations run slowly.

To minimize the size of the Petri nets and the state space, specific slicing methods are recommended. Unfortunately, though effective in a tiny hypothetical example, these slicing methods have little or no effect in real-world discrete-event systems. As a result, there is a need for alternate slicing approaches that are successful for Petri net models of massive real-world systems. As a remedy to the problem, Reggie Davidrajuh [3] presented a new Modular Petri net. Large Petri net models are divided into modules in a modular Petri net. These modules are small, and their state spaces are small enough that they may be thoroughly examined. The novel Petri net is implemented in the program GPenSIM [4], which is critical for modeling, analyzing, and optimizing real-world discrete-event systems with GPenSIM. Many advantages of modularization have been discovered, including flexibility (the ability to add or remove functionality), comprehensibility (model readability), reduced development time, and robustness (less prone to error). One of the new modular Petri net's development goals is to construct distributed systems; the modules must be able to execute on several processors. Modeling cyber-physical systems with geographically dispersed components but integrated via inter-modular communication are made possible by running modules on various computers.

All distributed systems have a communication infrastructure at their core. It's pointless to analyze distributed systems without looking at how processes on various computers might communicate data. In distributed systems, communication relied on low-level message forwarding provided by the underlying network. Because no distributed platforms exist, expressing communication through message passing is more complicated than utilizing primitives based on shared memory. When process A wishes to interact with process B, it creates a message in its own address space. Then it makes a system call, which instructs the operating system to transmit the message to B through the network. Although the underlying concept is straightforward, A and B must agree on the meaning of the bits being transferred to avoid unrest. To communicate via a network, a collection of computers must all agree on the protocols to be used. Before sharing data, the sender and receiver must explicitly establish a connection and perhaps negotiate the protocol they will use with connection-oriented protocols. Finally, they must disconnect the connection after they are finished. This thesis will communicate with Petri nodules using the TCP/IP protocol suite, which is the most widely used protocol. We will also examine the protocol's regulations.

In this project, two models of large discrete-event systems with Petri modules are designed: distributed systems for computing quadratic equations [5] and a client-server model of distributed systems. The thesis focuses on general infrastructure for communication between Petri modules. The modules that make up the model can be run on different computers. A network layer has been implemented in GPenSIM to allow the Petri modules to communicate with each other. The TCP/IP socket has been used for the communication between Petri modules.

1.1 Outline

There are six chapters in this thesis report. Chapter 2 discusses the technical and theoretical background in addition to related works. Chapter 3 provides the design and methods of the project where the block diagrams that explore the components of the system besides that the flowchart shows how the system is working. Chapter 4 represents the implementation of the systems of distributed modules. Chapter 5 provides the user manual and analyzes the simulation results. Finally, chapter 6 discusses the work's originality and limitations and proposes some future work that can be executed in the future.

Chapter 2

2 Background

This chapter provides a background to the work. This chapter has two sections. In the first section, the theoretical and technical background will be discussed, which is required to understand the work of this thesis. It starts by explaining the basic theory of Petri net and modular Petri net. Then, will give a brief idea about GPenSIM, a General-purpose Petri Net Simulator software used to implement modular Petri net and simulation purpose. Finally, it provides the basic knowledge of TCP/IP protocol and how it works in the MATLAB platform. The second section discusses some related works that have been done before on this topic. Then, it discusses the problem definition and approach solution to the problem.

2.1 Technical and Theoretical Background

As mentioned above, this section aims to talk about the technical and theoretical background of the topic. Therefore, this summary on technical and theoretical knowledge will help understand the work done in this thesis.

2.1.1 Petri Net

Because of its graphical representation and well-defined semantics, the Petri net is frequently used to model and simulate discrete-event systems [6]. It provides advantages such as being readily available, simple to comprehend, and simple to use. Petri nets have a distinct edge over other networks in terms of description and analysis. At the same time, a Petri net is a mathematical entity with a precise definition. It may be utilized for static structural analysis and dynamic behavior analysis thanks to the mathematical advancement of Petri nets analysis methodologies and techniques. Petri net modeling methodology may mimic systems having properties like concurrency, asynchrony, distributed parallelism, and uncertain equivalence.

Places and transitions are the two sorts of elements in a P/T Petri net. Transitions indicate active components, whereas locations represent passive components (such as input and output buffers and conveyor belts) (such as machines, robots). A Petri net is a directed bipartite graph, which means that a place may only be linked to transition(s) and a transition to place(s); arcs are the connections between places and transitions.

Petri nets have tokens in addition to places, transitions, and arcs. Tokens are things that may flow between nodes in a network, such as materials in a material flow system or data (or information) in an information flow. Tokens are stored in places, and they travel between them via the arcs. In a Petri net, tokens appear as black dots. When a place has a significant number of tokens, it is more common to use digits rather

than black dots to indicate the number of tokens. The default weight of the arcs connecting places to transitions and transitions to places is one. If the weight of an arc is more than unity, the weight is displayed in the arc. Thus, the arc weight is a measurement of the arc's ability to transfer many tokens at once.

P1, P2, and P3 are shown in Figure 1.1. These three places each have 5 tokens, 2 tokens, and 0 tokens. When a transition fires, several tokens are removed (‘consumed') from the input place, and new tokens are placed (‘produced') in the output place; the arc weights determine the quantity of tokens consumed and created. The number of tokens in the input places must be equal to or more than the weights of the arcs linking the input places to the transition for the transition to fire. After that, the transition will be able to fire (enabled transition). After the transition t1 has fired once, Figure 1.2 depicts the condition of the sample Petri net from Fig. 1.1.

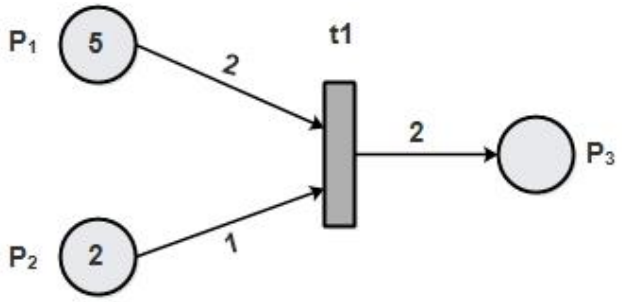

Fig. 2.1: Sample Petri net

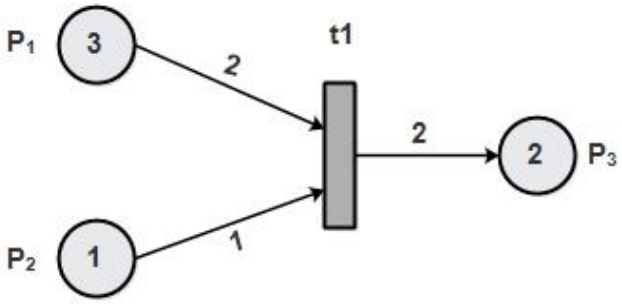

Fig. 2.2: Petri net after one firing of t1

In the Petri net shown in Fig. 1.2, the places P_1 and P_2 are input places to transition t_1, and P_3 is an output place of transition t_1.

- **Places:** Input and output buffers, as well as conveyor belts, are examples of passive components.
- **Transitions:** Active components such as machines and robots are represented by transitions.
- **Tokens:** Tokens are items that may move between nodes in a network.

Formal Definition of Petri nets

A Petri net is a 4-tuple (P, T, A, m_0)
Where

P is the set of places, ***P*** = {p_1, p_2, . . . p_n}
T is the set of transitions, ***T*** = {t_1, t_2, . . ., t_m}
A is the set of arcs (from places to transitions and from transitions to places)

$$A \_ (P \times T) \cup (T \times P), \text{ and } [6]$$

m is the row vector of markings (tokens) on the set of places
m = [*m* (P_1), *m* (P_1), . . ., *m* (P_n)] ∈ $\mathbf{N}^n$, m_0 is the initial marking.

2.1.2 Modular Petri net

There are certain drawbacks to using Petri nets to represent real-world discrete-event systems, such as the size of the model (even for a basic system), the slowness of simulation, the complexity of analyzing the model, and the explicit state space [3]. As a result, certain slicing methods are proposed for reducing the size of Petri nets and the state space. These slicing methods, however, are ineffective on real-world discrete-event systems.

The problem can be solved using a modular Petri net. Modularization is a way of breaking down Petri net models into modules to make them easier to develop and analyze. The huge and sophisticated Petri net models of real-life discrete-event systems are broken into modules in a modular Petri net. These modules are small, and their state spaces are small enough that they may be thoroughly examined. The following are some of the advantages of modularization:

- Adaptability (ability to add or change functionality)
- Comprehensibility (readability of the models)
- Reduced development time, as well as
- Robustness is a quality that describes a person's ability to withstand (less prone to error)

Numerous studies have been done on developing modular models with Petri nets, and many strategies, such as fusion places, fusion transitions, and substitution transitions, have been proposed. A modularization of Petri nets using fusion locations and fusion transitions was reported in ref. [20]. Fusion places and fusion transitions are two distinct types of locations and transitions. Due to the firings of the local (members of the module) transitions, these locations and transitions are only used to divide a Petri net model into modules and examine them independently. Though fusion locations and fusion transitions appear to be highly beneficial for modular model construction, they go against the core notion of modularization: data hiding.

A new modular Petri net was presented by Reggie Davidrajuh [3]. There are zero or more Petri Modules and Inter-Modular Connectors in a modular Petri net (IMC). The Petri modules are self-contained, allowing them to be built and tested separately. The Inter-Modular Connector (IMC) joins the modules together. This new modular Petri net was created specifically for use with GPenSIM.

The new design's objectives are as follows:

- **Data hiding:** The purpose of storing data inside modules is to abstract away internal information at a higher level.
- **Independent modules:** The modules are self-contained and have the potential to become self-contained.
- **Synchronization of modules:** Modules must be able to execute on separate processors, according to synchronization.

There are four separate sets of elements in a Petri module:

1. **Input ports:** The input gates of a module are input port transitions. Tokens can only be directed into the module through these transitions (input ports).
2. **Output ports:** The output gates of a module are output port transitions. Tokens can only be routed away from the module through these transitions (output ports).
3. **Local transitions:** A local transition consumes tokens from local input locations and deposits tokens into local output places as the module's local member (internal element).
4. **Local places:** A local location, as a module's local member, feeds tokens to the module's local transitions or input and output ports. Tokens are obtained by a local location from either local transitions or the module's input and output ports.

2.1.3 GPenSIM

GPenSIM defines a Petri net language for modeling and simulation of discrete-event systems on MATLAB platform [6]. GPenSIM is developed by Reggie Davidrajuh [4]. GPenSIM is also a simulator with which Petri net models can be developed, simulated, and analyzed. In addition, GPenSIM can also be used as a real-time controller. Even though GPenSIM is a new simulator, many researchers around the world are using it. GPenSIM is easy to learn, use, and extend.

With the addition of the ability to create modular Petri net models, GPenSIM may now be used to model and evaluate real-world systems. The newer version of GPenSIM (version 10) allows modularization so that flexibility (ability to add or change functionality) and comprehensibility (readability) of the model can be improved. GPenSIM supports many Petri nets extensions, such as inhibitor arcs, transition priorities, enabling functions, color extension. In addition, it provides a collection of functions for performance analysis. Because of its flexibility, it is also easy to implement any other Petri net extensions with GPenSIM

GPenSIM was created with three primary objectives in mind:

1. Discrete-event systems modeling, simulation, performance analysis, and control.
2. A tool that is simple to use and expand, as well as
3. Integration of Petri net models with other MATLAB toolboxes.

Methodology for Creating a Petri net with GPenSIM

There are three steps to create a Petri net model in GPensIM-

Step-1. Defining the Petri net graph in a PDF.
Step-2. Coding the firing conditions in the relevant pre-processor files and post-firing activities in the post-processor files.
Step-3. Assigning the initial dynamics of a Petri net in the MSF.

- **Petri net Definition Files (PDF):** The Petri net Definition File contains the definition of a Petri net graph (static information). The purpose of a PDF is to identify the elements (places, transitions) of a Petri net and specify how they are connected. If the Petri net model is broken into several modules, each module is defined in its PDF; there may be several PDFs.

- **Pre-processor file:** A pre-processor file contains the code for additional conditions to check whether an enabled transition can fire; in other words, a pre-processor is run before firing a transition to make sure that an enabled transition can start firing depending upon some other additional conditions ('firing conditions'). Further, we can write separate pre-processors for each transition or combine them into a single common pre-processor. It is also allowed to use individual pre-processors together with the common pre-processor.

- **Post-processor file:** A Post-processor file is run after the firing of a transition. A post-processor contains code for actions that have to be carried out after a specific transition completes firing. Just like pre-processors, post-processors can be specified for individual transitions or combined into one common post-processor.

- **Main Simulation File (MSF):** Main simulation file contains the dynamic information such as initial tokens in places, firing times of transitions of the Petri net.

Integrating with MATLAB Environment

One of the most important reasons for developing GPenSIM and its most advantage is its integration with the MATLAB environment to harness diverse toolboxes available in the MATLAB environment [6]; see Fig. 2.2. For example, by combining GPenSIM with the Control System Toolbox, we can experiment with hybrid discrete-continuous control applications.

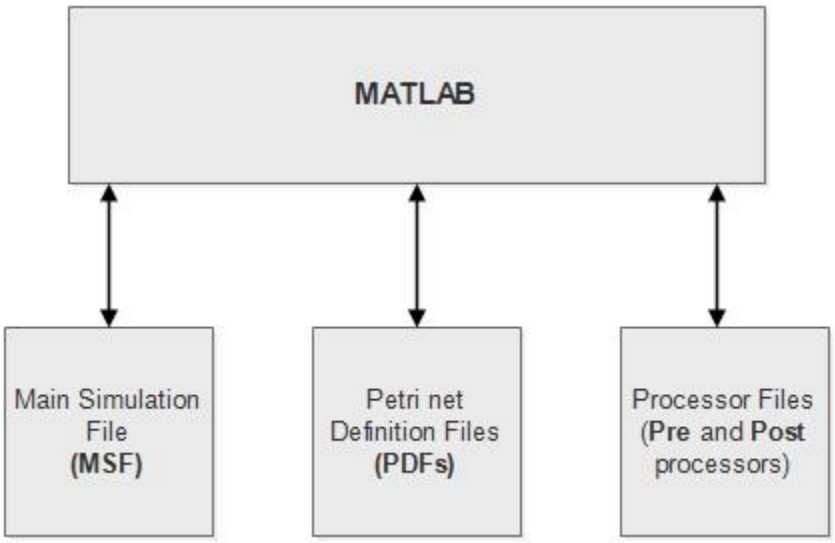

Fig. 2.3: Integrating with MATLAB environment

2.1.4 TCP/IP

Transmission Control Protocol/Internet Protocol (TCP/IP) allows digital machines to interact across great distances. The Internet protocol suite specifies how data should be packetized, addressed, transferred, routed, and received from beginning to finish.

This functionality is divided into four abstraction levels, each of which categorizes all connected protocols based on the breadth of the networking involved. The layers are as follows:

1. **Application layer:** Responsible for node-to-node communication and controls user-interface specifications.

2. **Transport layer:** Responsible for end-to-end communication and error free delivery of data.

3. **Internet layer:** Provides internetworking between independent networks.

4. **Network access/Link layer:** Contains communication methods for data that remains within a single network segment

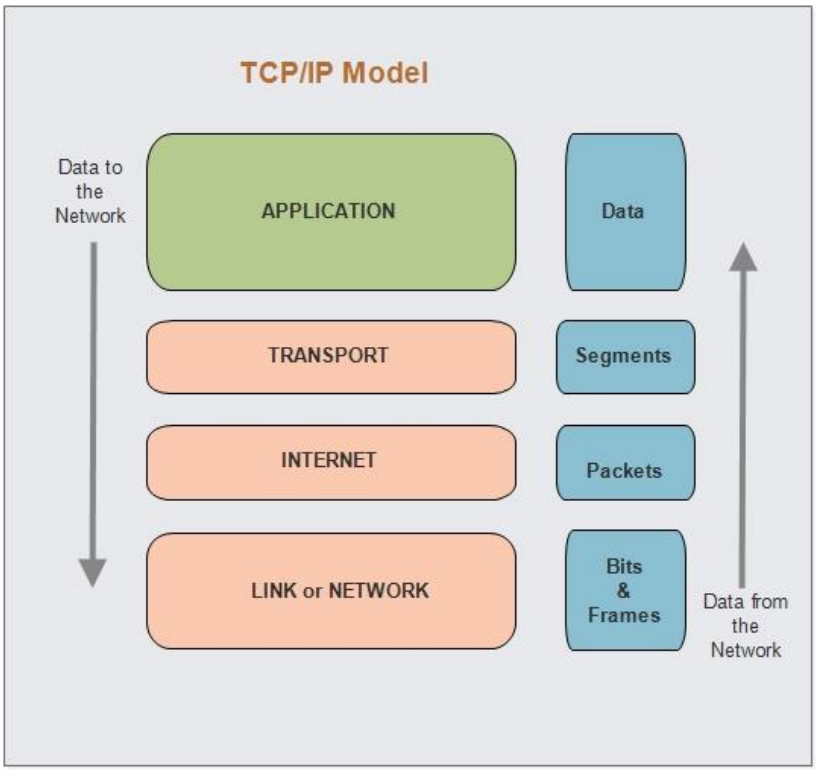

Fig. 2.4: TCP/IP model

TCP/IP in Matlab

TCP/IP communication functionality in Instrument Control Toolbox offers the ability to construct clients and servers. TCP/IP client functionality allows you to connect to distant hosts via network socket communication from MATLAB for reading and writing binary and ASCII data. In addition, you may construct a network socket for communication between MATLAB and a single client using TCP/IP server functionality.

TCP/IP communication with a Remote Host

There are some steps required to communicate with a remote host using TCP/IP.

1. **Create an instrument object-** First need to create a TCP/IP instrument object.
   t = tcpip ('127.0.0.1', 80);
   Here, 127.0.0.1 is the host address and 80 is port. In most cases, you need to specify the value of the port otherwise 80 will be used as default value.
2. **Connect to the instrument-** Once the object is created, you need to establish a connection with remote host by open the connection to the server. The function is-
   fopen (t);
   An error may occurs if the server is not available, busy or do not accept the connection request.

3. **Write and read data-** If the connection established successfully, you will be able to communicate with the server. The functions of write and read operations are- *fprintf, fscanf, fwrite and fread*.
4. **Disconnect-** If the write and read operations are done, disconnect the communication with the object by calling *fclose* function.
   fclose (t);

TCP/IP Server Sockets Communication

TCP/IP Server Sockets Communication in Matlab is available using the *tcpserver* function. This support is for a single remote connection. This connection is used to communicate between a client and MATLAB or between two instances of MATLAB. For example, collect data from one instance of MATLAB and then transfer it to another instance of MATLAB.

In server socket communication, it is necessary to set *the NetworkRole* property in the *tcpip* interface. It uses two values, *client* and *server,* to establish a connection as the client or the server. The server socket feature supports binary and ASCII transfers.

There is an issue with TCP/IP server socket communication in MATLAB; while a server socket is waiting for a connection after calling *fopen,* the MATLAB processing thread is blocked.

Communication between Two Instances of MATLAB

The sample code for each session is shown in the following example, which explains how to link two MATLAB sessions on the same machine. In the code for Session 2, substitute "localhost" with the server's IP address to utilize two distinct PCs. The IP address "0.0.0.0" indicates that the server will accept the first computer that attempts to connect. Replace "0.0.0.0" in the code for Session 1 with the client's address to limit the connections that will be made.

Session 1: Server

A TCP/IP server is represented by a tcpserver object, which accepts a TCP/IP client connection request from the supplied IP address and port number. The data may be obtained from the client using the read function once the server has established a connection.

```
t = tcpip ('0.0.0.0', 4000, 'NetworkRole', 'server');
fopen(t);
fread(t);
```

The server accepts a connection from any machine on port 4000 and open the connection by calling *fopen(),* then read the data by function *fread()*.

Session 2: Client

A tcpclient object in MATLAB represents a connection to a remote host and port for writing and reading data. The remote host must already exist and can be a server or hardware that supports TCP/IP connection.

```
t = tcpip ('localhost', 4000, 'NetworkRole', 'client');
fopen(t);
fwrite(t, data);
```

Client, creates a client interface and looks for connection on port 4000 by *fopen().* Then, writes data to the server by calling *fwrite()* function.

Functions and Properties to Read and Write Data over TCP/IP

These are some functions used to read and write ASCII and binary data over TCP/IP.

Functions	**Purpose**	**Operate mode**
fprintf	Writes data as text to the server	Synchronous
fscanf	Reads data as text from the server	Synchronous
fwrite	Writes binary data to the server	Synchronous
fread	Reads binary data from the server	Synchronous
fgetl	Reads a line from the server	Synchronous

There are some properties, which are associated with reading and writing operation of data over TCP/IP.

Property	**Purpose**
InputBufferSize	Specify the size of input buffer in bytes during read operation
OutputBufferSize	Specify the size of output buffer in bytes during write operation
Timeout	States the waiting time to complete read and write operations in second
ValuesReceived	Total number of values read from the server
ValuesSent	Total number of values written to the server

When the object operates in synchronous mode, the MATLAB command line is blocked by the reading and write routines until the job is done or a timeout occurs. When the object operates in asynchronous mode, the read and writes routines return control immediately to the MATLAB command line. Read data asynchronously by setting the function R*eadAsyncMode* in continuous or manual mode.

```
t.ReadAsyncMode = continuous;
```

2.2 Literature Review and Formulation of the Problem

This section aims to discuss some related works. First, I will give a short literature review on some of the many beautiful pieces that have been done about Petri net. Then, I will define the problem going to be solved, explaining why the topic is engaging.

2.2.1 Literature review

The basic P/T Petri net does not support modules. Savi [7] is one of the early works to mention Petri Net modules. Savi (1992) used event graphs as modules, and the modules had transitions as interfaces.

De (1994) also is one of the early works on modules [8]. De (1994) focuses on modeling and optimization of circuit boards [8]. Proposed the modular Petri net solution to the problem. Here, the proposed model must be designed so that all the necessary communication must be preserved in one module to avoid complexity and improve efficiency. They have also suggested using a two-level hierarchy in the models for better and fast design.
Xue (1998) focuses on modeling flexible manufacturing systems with modular Petri net models [9]. In Xue (1998), a flexible manufacturing system is divided into subsystems (e.g., the arrival of raw material, machining, and finishing, etc.), and each subsystem is modeled as a Petri net module [8].

After that, the fusion modular-Petri net gain importance. Refs [10], proposed a model using the fusion places in place of standard places and the fusion transitions in place of standard transitions. These places and transitions are modified versions of the original that are used to convert the models in modules based on firing of the local transitions.

Tsinarakis (2005) is about reusable generic modules [11]. Tsinarakis (2005) assumes that a manufacturing system can be divided into basic building blocks such as production lines, assembly, disassembly, and parallel machining elements. And these basic building blocks can be developed as generic modules, then customized to specific needs [11]. Similar to Tsinarakis (2005), Lee (2009) also proposes "reconfigurable modules'' to tackle uncertainties associated with models [12].

The modular Petri net modules are also used in the medical modeling and applications. Refs [13, 14], proposed a modular-Petri net system for the “Spanish Health System”. The proposed model used the advantages of the modular scheme and model, the complex system in an easy systematic fashion. The model allows the easy analysis and exploration of the system.

There are tools for making modular Petri net models. Bonnet (2006) presents "Exhost-PIPE'' [15], and Jensen (2015) present "CPN", a well-known tool for Petri Nets [16]. Though all these tools support modular model building, they do not support distributed modules that can communicate between themselves.

Refs [17], proposed a modular network for the avoidance of risk or threats in an IT system. Here, two different set of models have been used for the design of an overall system. One of them is used for the production systems and other for the IT system. The overall system is referred to as information and the control network.

The idea of hosting Petri modules on different computers that are geographically kept apart and the modules communicating between themselves did not exist before it was presented by Davidrajuh (2020) [18]. Davidrajuh (2020) presents geographically distributed Petri modules to minimize the simulation time, reduce the state space size and other complexities, and keep the modules closer to where they are needed [18]. To develop distributed Petri net modules, Davidrajuh (2019) presents a new modular Petri Net with well-defined Petri modules [3]. All the ideas behind the distributed Petri modules are implemented in the software known as General-purpose Petri Net Simulator (GPenSIM), presented in Davidrajuh (2018) [6].

2.2.2 Problem definition

Modeling a large discrete-event system with Petri modules in which the modules are distributed. In a large modular Petri net model, the modules that make up the model can be run on different computers. We need to make a network layer in GPenSIM, allowing the Petri modules to communicate with each other using TCP/IP sockets.

Many works have been done on Petri net and modular Petri net, but these are not distributed modules, and communicating ideas do not exist between modules. The proposed idea [11], distributed Petri modules are different and quite interesting because modules must be running on different computers and exchange information between them through a communication channel. This master thesis can be considered as a continuation of the work presented in Davidrajuh (2020). Though Davidrajuh (2020) presents the idea and definition of distributed modules, the communication between modules is not mentioned in detail.

In this thesis, two systems developed with modular Petri net where system possesses different modules, the modules are distributed and capable of running on different computers. The modules are using the TCP/IP for communication. TCP/IP is the Internet protocol suite available with every computer; thus, it is chosen as the protocol for data exchange between the modules.

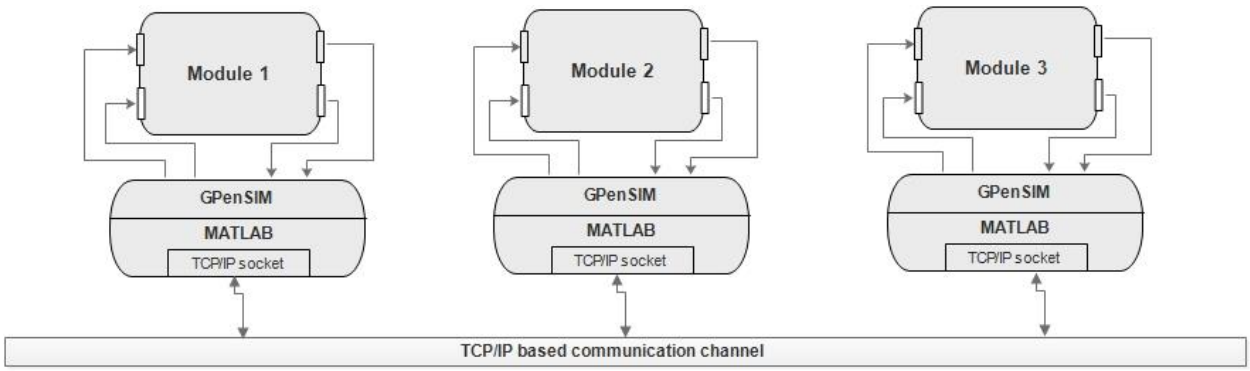

Fig. 2.5: Distributed Petri modules

Fig. 2.5 shows a system consists of three modules, where the modules are distributed and connected with a TCP/IP-based communication channel to exchange information between them. A network layer is implemented in GPenSIM using MATLAB TCP/IP sockets so that the distributed agents can communicate with each other and exchange information to perform the job.

The models are designed and implemented with GPenSIM. This modeling approach minimizes the simulation time and reduces the state space size and other complexities. In addition, GPenSIM is used to analyze and understand the performance of the system.

Chapter 3

3 Method and Design

This chapter discusses the design architecture of the models and the techniques are used to design the modules. It starts with discussing the design of a distributed system for computing quadratic equations and the overall functions of the system. Then, discuss the design of the client-server model to dissolve overall into modules and necessary operations performed by transitions and places of the modules. Finally, it discusses the methods/techniques are used to implement the distributed systems for communication.

3.1 Design

In this thesis, two modular Petri net models are designed and implemented in GpenSIM. The idea of creating the modular Petri net is to make the Petri modules enable to run on different computers and communicate with each other over the network. One model is developed for a system to compute a quadratic function; the system consists of three modules: client, multiplier, and adder. Another model is designed for client-server communication; this system also possesses three modules. In both systems, the modules are distributed and can be run on different computers and can communicate in a distributed environment. The overall design of the two systems is explained below.

3.1.1 Distributed System for Computing Quadratic Equation

The system consists of three distributed modules (communicating agents), client, multiplier, and adder to solve a quadratic function (e.g., $f = ax^2 + bx + c$) collaboratively.

- **Client:** takes the values of the variables a, b, c, and x, and provides the job to be done.
- **Multiplier:** receive the input values of a, b, c, and x, and do the multiplication. E.g, perform (*a,x,x*), (*b,x*) and returns the calculated values to the client.
- **Adder:** takes the input value, perform arithmetic operations, and return the arithmetic sum. E.g, takes input (*a,b,c*) and returns (*a+b+c*).

Fig. 3.1 shows the sequential operations between the modules. The client, multiplier, and adder are three distributed Petri modules; they exchange information like TCP/IP packets to complete the job. In fig. 3.1, the client first sends the values of (*a.b,c,x*) to multiplier as input. After multiplication is done, the multiplier returns the result values to the client. E.g., (*a.x2, b.x, c*). Then, the client receives the result

values and sends them to the adder as input. Finally, the adder returns the arithmetic sum to the client as the final result, and the job is done.

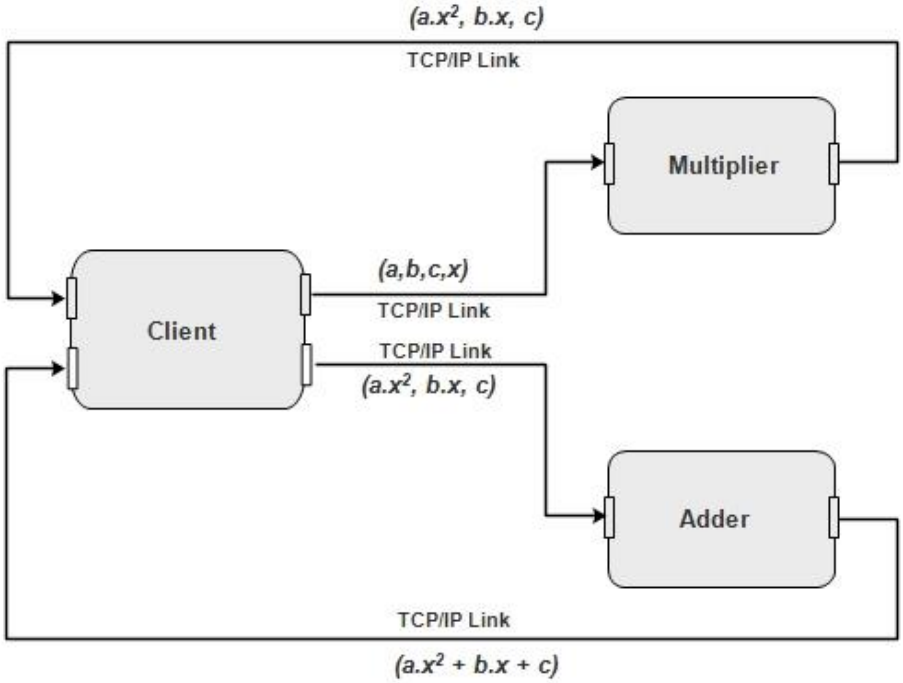

Fig. 3.1: Basic functionality of the modules

Overall Design of the Model

Each module of the system consists of transitions and places, where the transitions represent the activities and places represent the buffer for messages [3]. For example, fig. 3.2 represents an IO port-driven-based modular Petri net model designed and implemented in GPenSIM. Three Petri modules (client, multiplier, and adder) make up the Petri model for computing quadratic equations connected by a TCP/IP-based communication channel. When the modules are distributed that need to communicate, a set of transitions is defined as I/O ports through which the modules can send their packet to communicate with each other.

Client: has four transitions and four places, two input transitions (***tCI_Mul, tCI_Add***), two output transitions (***tCO_Mul, tCO_Add***), and four places (***pC0, pC1, pC2, pC3***) respectively.

Multiplier: consists of three transitions and two places, one input (***tMI***), one output (***tMO***), one local transition (***tM***), and two places (***pM1, pM2***) respectively.

Adder: has same as multiplier, three transitions and two places. One input (***tAI***), one output (***tAO***), one local transition (***tA***), and two places (***pA1, pA2***) respectively.

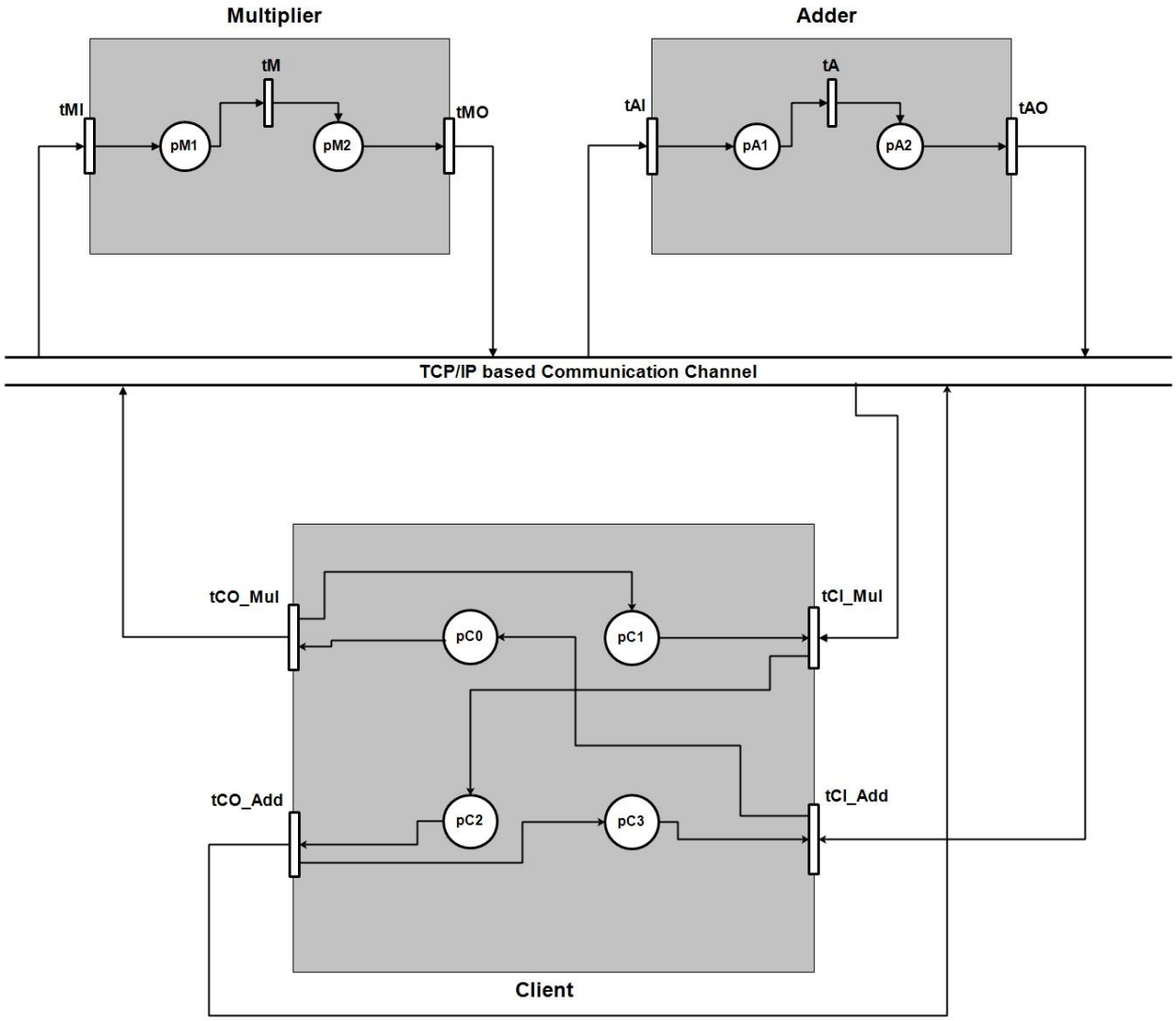

Fig. 3.2: IO port-driven based modular Petri model

Functionalities of the System

To solve the quadratic function (e.g., $f = ax2 + bx + c$), the transition ***tCO_Mul*** takes input values of the respective variables and creates a message as TCP/IP packet. After making the packet, it requests a connection to the multiplier server to send the packet/message (values) through TCP/IP socket, ***tCO_Mul*** acts as a client.

In the multiplier module, the input transition ***tMI*** acts as a server that establishes a connection when it gets the connection request from the client and reads the data packet. Once the message is received, the values (message/packet) are saved to place ***pM1*** as a token. Transition ***tM*** is a local transition, which consumes the token from the place ***pM1,*** performs computation (multiplication), and saves the result to the place ***pM2*** as a token. The output transition ***tMO*** acts as a module client, which consumes the token

from place ***pM2*** and creates a TCP/IP packet message. Then, ***tMO*** sends a connection request to the server of the client module to return the result values through the communication channel.

In the client module, the ***tCl_Mul*** is an input transition that serves as a server of the client module that establishes a connection when it gets connection requests from the multiplier module and starts reading the data packet. The received values (message/packet) are saved into the place ***pC2*** as a token. Transition ***tCO_Add*** consumes the token from the place ***pC2*** and creates a data packet to send it to the adder module for arithmetic operations. The ***tCO_Add*** acts as a client, which requests a connection to the adder server and sends the packet through TCP/IP socket.

The functions of the adder module are the same as the functions of the multiplier module; only the local transition ***tA*** is responsible for doing the arithmetic operations instead of multiplication and save the sum into the place ***pA2*** as a token. Finally, ***tCl_Add*** establish a connection after getting a connection request from the adder and read the packet to receive the results as a message, ***tCl_Add*** acting as a server of the client module. Display the results, and the job is done. After one job is done, the client gets ready for another job or computation.

Table 3.1, 3.2, and 3.3 explain the functions of the transitions and places of the different modules.

Elements	Purpose
tCO_Mul	Gets input values, makes connection request, and send values to the multiplier as TCP/IP packet.
tCl_Mul	Establish connection with multiplier, reads network packet, and receive messages from multiplier.
tCO_Add	Consumes token (message/values) from input place, make connection request, and send the values to the adder.
tCl_Add	Establish connection with adder, reads data packet, and receive the results from adder.
pC0, pC1, pC3	These three places used only to make connection with transitions into client module for looping purpose
pC2	The result values from multiplier is saved into this place.

Table 3.1: Elements of the client module and their purpose

Elements	Purpose
tMI	Establish connection with client, reads network message, and save the values into respective place.
tM	Do the necessary multiplication.
tMO	Consumes tokens from input place, creates data packet, and make connection request to returns the results.
pM1	The values from client are saved into this place.
pM2	Calculated values of local transition **tM** are saved into this place.

Table 3.2: Elements of the multiplier module and their purpose

Elements	Purpose
tAI	Establish connection with client, reads network message, and save the values into respective place.
tA	Do the arithmetic operations.
tAO	Consumes tokens from input place, creates data packet, and make connection request to returns the results.
pA1	The values from client are saved into this place.
pA2	Arithmetic values calculated by **tA** are saved into this place.

Table 3.3: Elements of the adder module and their purpose

3.1.2 Client-Server Model

The client-server model is designed to see how the communication of distributed systems works in client-server interaction patterns. In this thesis, TCP/IP is used as middleware for communication in a client-server architecture. The system can be considered an example of an electronic mail system where every user agent is allowed to write, send, and receive e-mail. In this modular Petri net model, when a module wants to communicate with another module, it first creates a message in its address, passes the message to the server, and then gets acknowledgment that message is delivered. Likewise, the receiver module connects to the server and check whether any message is addressed to it. If so, read the message and display it. Fig. 3.3 shows the basic client-server interaction.

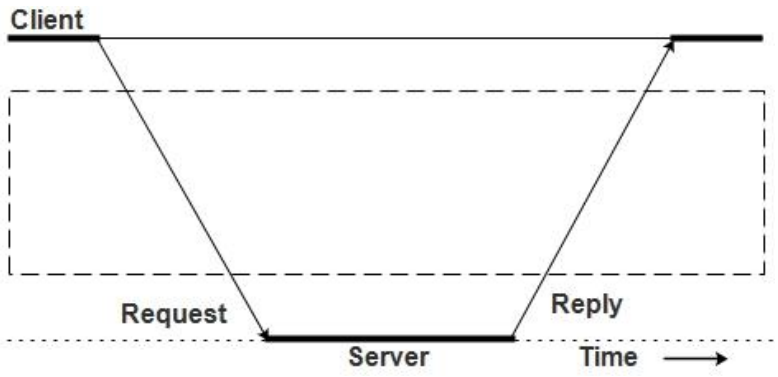

Fig. 3.3: The client-server interaction

Overall design of the system

The system contains three different Petri modules in which the modules are distributed and run on different processors. In addition, these modules exchange information like TCP/IP packets. Here, modules **A** and **B** are communicating agents, allowed to send and receive messages. The module **X** acting as a server, I would say a "transceiver" because this module has a direct TCP/IP-based connection to **A** and **B** to receive and transfer the message for communication between **A** and **B**.

- **Module A:** Generates tokens (messages) at different times and sends the message to module **B** through the module **X**. In addition, the module receives messages addressed to it. There is no connection or provision to send any message to module **B** directly.

- **Module B:** Generates tokens (messages) at different times and sends the message to module **A** through module **X**. Moreover, the module receives messages addressed to it. There is no connection or provision to send any message to module **A** directly.

- **Module X:** Acts as a transceiver. Therefore, **X** receives messages from **A** and **B**, checks the source and destination, and then transfers the messages to the destination.

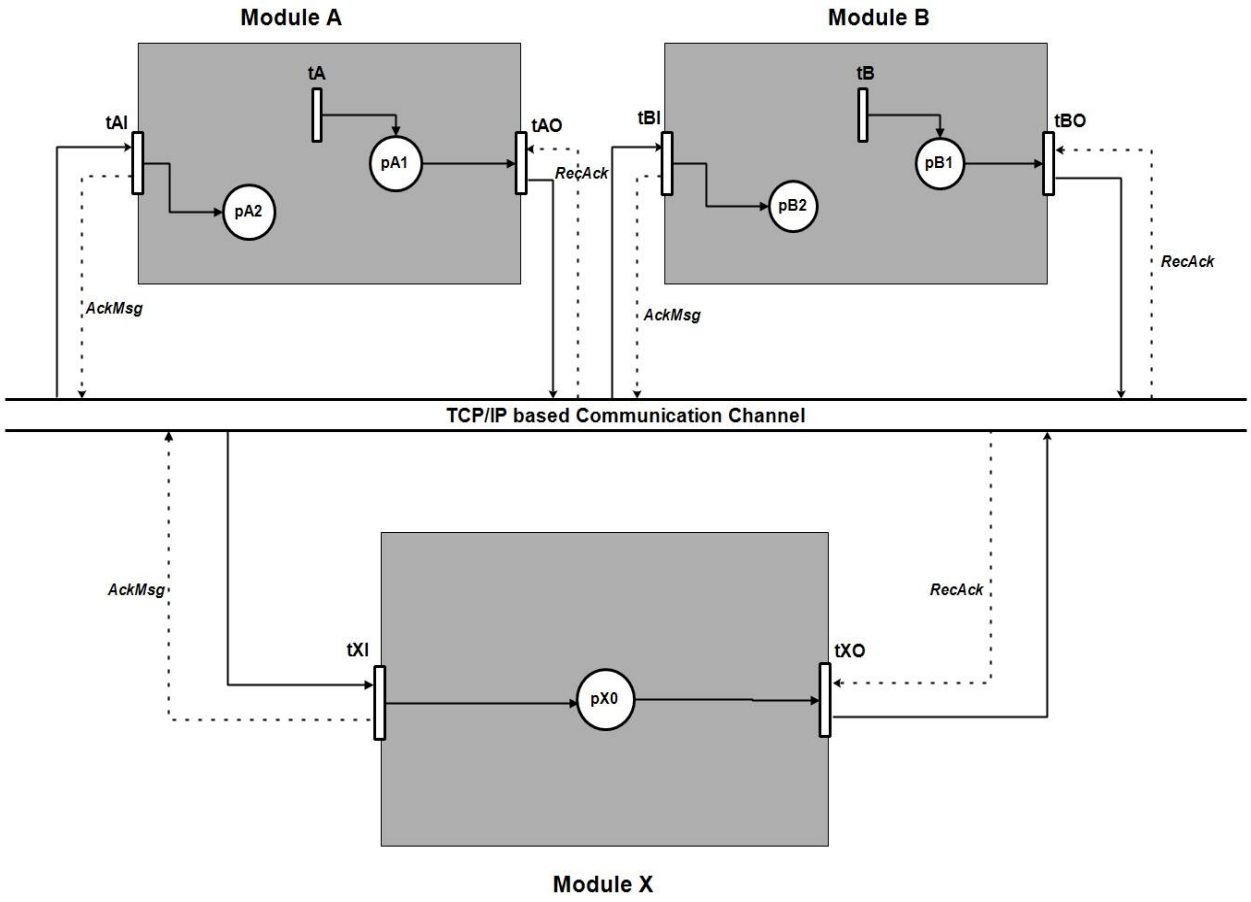

Fig. 3.4: Distributed system in client-server architecture

Functionalities of the system

Fig. 3.4 represents a client-server based distributed system, which is designed and implemented in GPenSIM. A set of transitions are defined as I/O ports through which the modules can communicate with each other.

Module A has three transitions and two places. One input transition (***tAI***), one output transition (***tAO***), one local transition (***tA***), and two places (***pA1 and pA2***).

Module B: contains three transitions and two places. One input transition (***tBI***), one output transition (***tBO***), one local transition (***tB***), and two places (***pB1 and pB2***).

Module X: consists of two transitions and one place. One input transition (***tXI***), one output transition (***tXO***), and a place (***pX0***).

Tokens (messages) are generated by the local transitions of both modules **A** and **B** arbitrarily. For example, suppose in module **A**, the local transition ***tA*** generates a token at time unit 20 and saved into place ***pA1***. The token possesses two colors, "From: A" and "To: B". ***tAO*** consumes the token in

place ***pA1*** and creates a message (data packet). Then, ***tAO*** requests for a connection to server **X** to send the message. Also, ***tAO*** receives an acknowledgment from the server as the message is received.

In module **X, *tXI*** will establish the connection request and reads the network message to receive it. ***tXI*** also sends an acknowledgment to the sender. The received message will be stored in place ***pX0*** as a token. The ***tXO*** consumes the token from the place ***pX0*** and extracts the message by identifying the source and destination. For example, if the message's source is "From: A", it will make a connection to the destination (e.g., To: B) of the message to transfer the message to the intended destination and vice versa. ***tXO*** will get an acknowledgment once the recipient receives the message.

Transition ***tBI*** of module **B** will listen to the communication channel and establish the requested connection of module **X.** ***tBI*** reads the network message once the connection is established, send an acknowledgment message, and saves the message to the place ***pB2.*** The transition ***tAI*** of module **A** will do the same function as of module **B.**

The activities of module **B** are similar to **A.** The only difference is that when local transition ***tB*** generates tokens, the colors would be "From: B" and "To: A". This is how modules A and B will generate tokens at different times and communicate with each other through module X.

Elements	Purpose
tXI	Establish the connections, removes messages from TCP/IP, sends acknowledgment message, and store into *pX0* as a token
tXO	Consumes the token from *pX0,* identify the source and destination of the message; make connection to the destination to transfer the message, and receive acknowledgment once the message is received.
pX0	Holds the received messages by *tXI.*

Table 3.4: Transitions and places of module X and their purpose

Table 3.4, 3.5, and 3.6 represents the transitions and places of the modules and their functionality.

Elements	Purpose
tAI	Listen to the communication channel; receive any message addressed to it, send an acknowledgment message to the sender, and save the received message as a token in *pA2.*
tA	Produces tokens arbitrarily at different times and deposit the tokens into *pA1.* Tokens possess two colors “From: A” and “To: B”
tAO	Consumes the token in place *pA1*, creates data packet, sends a connection request to X server, and receives acknowledgment.
pA1	Holds the token from *tA*
pA2	Holds the received message by *tAI*

Table 3.5: Transitions and places of module A and their purpose

Elements	Purpose
tBI	Listen to the communication channel; receive any message addressed to it, send acknowledgment message, and save the received message as a token in *pB2.*
tB	Produces tokens arbitrarily at different times and deposit the tokens into *pB1.* Tokens possess two colors “From: B” and “To: A”
tBO	Consumes the token in place *pB1*, creates data packet, sends a connection request to X server, and get acknowledgment.
pB1	Holds the token from *tB*
pB2	Holds the received message by *tBI*

Table 3.6: Transitions and places of module B and their purpose

Fig. 3.5 shows how the modules work during the system running time. When the system started, three modules are running. Module X has direct TCP/IP-based connections to A and B to receive and transfer the messages. After generating the tokens by modules A and B, send a connection request to X, TCP/IP; check the connection status if connections status is open and the server is ready, then establish the connection with the server. If the connection status is closed or the server is busy with another job, it will wait for a while and send a connection request again. Then, X receives the tokens and extracts the message, check all the possible statements to make the connection with the intended destination, e.g., if the message is from **B**, then send the message to **A** or if the message is from **A** then send the message to **B**. After the job is done, the server will be activated again to receive another message.

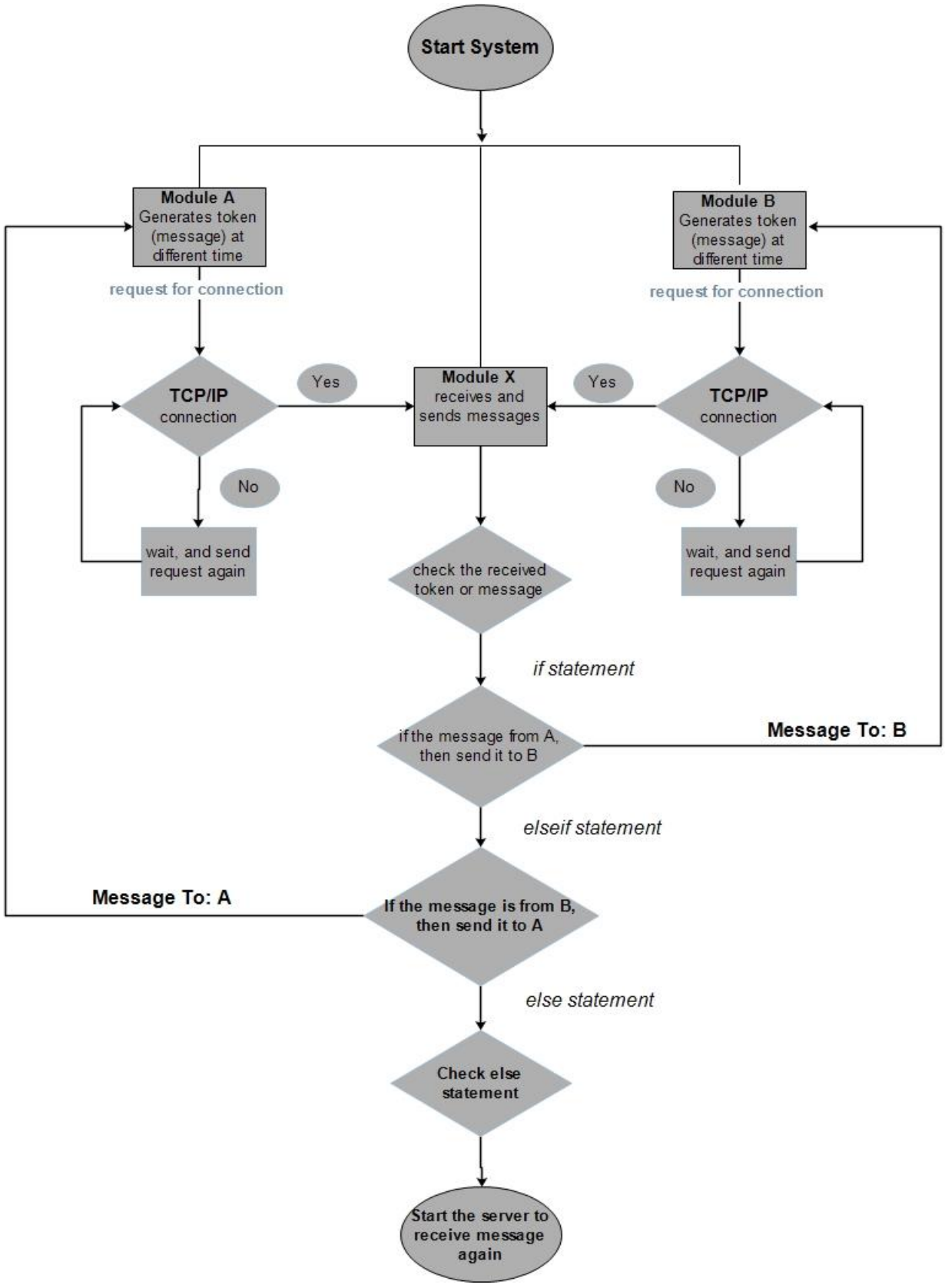

Fig. 3.5: Executions of the modules in client-server based distributed system

3.2 Techniques

This section discusses the techniques and methods are used to implement the modular Petri models and the network protocol used to developed communication between the modules. There are two techniques used to model the Petri Modules and the TCP/IP protocol used to implement the communication between the modules.

3.2.1 IO port-driven Modules

The GPenSIM supports three types of modules: IO port-driven, IO buffer-driven, and Hybrid type module. In IO port-driven modules, the input and the output gates of the module are transitions. At the same time, the input and the output gates of the module are placed in an IO buffer-driven module.

IO port-driven modules for modeling more independent agents in a peer-to-peer topology, whereas IO buffer-driven modules for modeling agents in a master-slave environment. The models are designed as IO port-driven modules as it is better for the peer-to-peer approach, where the transitions represent the activities, and the places represent buffer for messages. All the messages exchanged between the modules happens in the form of tokens. In IO port-driven modules, each module can take the tokens (jobs) destined for them, perform the work, and then send the results to whoever needs the results through the com-mon buffer. However, IO port-driven modules need more implementation (coding) as more functions have to be programmed for grabbing the jobs from the common buffer and placing the results back into the common buffer with the address of the recipient tagged on the results.

3.2.2 Colored Petri Net

GPenSIM's coloring is fundamental, as it only accepts ASCII tags as colors [3]. Colored Petri Net (CPN) is a mathematical construct based on Petri nets. Coloured Petri nets maintain valuable Petri net traits while also extending the basic formalism to allow token differentiation. Colored Petri Nets vary from standard Petri Nets in one crucial way: tokens are more than blank markers; they include data. Tokens can have a data value linked to them in CPN. The token color refers to the connected data value. Even though the color can be of any kind, colored Petri nets usually contain tokens of just one kind. The place's color set is the name for this category. The modeling language can function with systems that require synchronization, communication, and resource sharing thanks to the data tokens.

CPNs are extremely useful for conversational modeling because:

- They're a straightforward formal model.
- They are shown graphically.
- They enable concurrency, which is required for many complex interactions.
- They've been well investigated and understood, and they've been used in a variety of real-world scenarios.

- There are several tools and methodologies for designing and analyzing CPN-based systems.

Coloring Tokens

- When using colors in GPenSIM:
  - **Only transitions can manipulate colors:** in Pre-processor, one can add, delete, or alter colors of the output tokens.
  - **By default, colors are inherited**: when a transition fires, colors from the input tokens are collected and given to the output tokens. However, color inheritance can be prevented by **overriding.**
- Enabled transition can select **specific input tokens** based on **colors.**
- Enabled transition can also select **specific input tokens** based on **time**; e.g. the time the tokens are created.
- Structure of tokens: tokens have a unique **tokID** number, in addition, **creation time**, and a **set of colors**.
- In GPenSIM, color of a token is a set of strings on the token.

3.2.3 TCP/IP Socket Communication

The TCP/IP socket server communication is used to developed communication between the modules. A socket is a communication endpoint where the read and write operation can be performed to receive and send data over a network.

TCP/IP used some socket primitive for the communication purpose, to create a new communication socket primitive is called. Bind primitive is used to bind the IP address together with the port number to a socket. Listen primitive is the willingness to accept connections. Accept primitive block the caller until a connection request arrives. Send and receive primitives are used to sending and receiving data over a connection, respectively. Close primitive release the connection.

The client end first creates a socket calling socket primitive but doesn't need to use bind and listen primitive. Moreover, the client uses connect primitive instead of listening primitive to specify the address to which the connection request to be sent. The rest of the primitives are the same for both the client and server end.

Chapter 4

4 Implementation

This chapter explains how each module of the system is implemented to discuss how the input, output, and local transition work. Moreover, how the connections are developed as both client and server will be discussed in this chapter. Section 4.1 explain details how each module of the model function, with necessary codes. Finally, section 4.2 describes the detailed implementation of each module of the client-server model.

4.1 Distributed System for Computing Quadratic Equation

This model is designed and implemented to compute a quadratic function. The system consists of three distributed Petri modules and communicates through TCP/IP. Section 3.1.1 explained the overall design and purpose of these three modules, how many transitions and places are in each module, and their functions. In addition, a brief explanation was given in that section about how the whole system works and communicates to complete the quadratic function. This section aims to discuss how each module is implemented.

4.1.1 Client

Client module contains six MATLAB files, Main Simulation File (MSF), Petri net Definition File (PDF), and four pre_processor files. In PDF, the transitions and places are identified, and defined how they are connected. The pre_processor files have code, written the conditions how an enable transitions can fire.

```
function [png] = module_client_pdf()
png.PN_name = 'Client';
png.set_of_Ps = {'pC0', 'pC1', 'pC2', 'pC3'};
png.set_of_Ts = {'tCO_Mul', 'tCI_Mul', ...
                 'tCO_Add', 'tCI_Add'};
png.set_of_As = {
                 'pC0','tCO_Mul',1, ...
                 'tCO_Mul','pC1',1, ...
                 'pC1','tCI_Mul',1, ...
                 'tCI_Mul','pC2',1, ...
                 'pC2','tCO_Add',1, ...
                 'tCO_Add','pC3',1, ...
                 'pC3','tCI_Add',1, ...
                 'tCI_Add','pC0',1};
png.set_of_Ports = {'tCO_Mul', 'tCI_Mul', 'tCO_Add', 'tCI_Add'};
```

The module has two input ports ('tCI_Add','tCI_Mul') and two output ports ('tCO_Add', 'tCO_Mul'), ports are responsible to send and receive data when module communicate in distributed environment.

```
disp('Kindly Enter the values of the variables: ')
a = input('Enter the value of a: ');
b = input('Enter the value of b: ');
c = input('Enter the value of c: ');
x = input('Enter the value of x: ');

if ((a == 0) && (b == 0) && (c == 0) && (x == 0))
    global_info.STOP_SIMULATION = 1;
else
    %constructing the client message

client_message=char([num2str(a),',',num2str(b),',',num2str(c),',',nu
m2str(x)]);
```

When we run the client module, transition `tCO_Mul` prompts user input for the variables of (*a,b,c, and x*), and checks the condition statements. If the values of all the variables are "0" then it stops the simulation. Otherwise, the client module takes multiple input and run a loop until the variables are all "0".

```
client_mul = tcpip('localhost',7000,'NetworkRole','client');
fopen(client_mul);
fwrite(client_mul,client_message);
```

If the else statement is true, it creates a message (data packet), requests a TCP/IP connection, and writes the message to the network to send it to multiplier through TCP/IP. Here, the transition is acting as a client; from the code, we can see the network role is a client.

```
client_server2 = tcpip('0.0.0.0',4000,'NetworkRole','server');
fopen(client_server2);
read_msg_from_mult = fgetl(client_server2);
```

After firing transition **`tCO_Mul`**, transition **`tCI_Mul`** open the server and establish connection with multiplier to get the result (ax^2*, b.x, and c*) back. From the code, we can see the network role is "server". Read the network message by calling the function "fgetl", and extract the message after receiving.

```
msg=[num2str(received_msg(1)),',',num2str(received_msg(2)),',',num2s
tr(received_msg(3))];
transition.override = 1;
transition.new_color={msg};
```

Then, stores the received message in place pC2.

Next step is to send the values of (ax^2, *b.x, and c*) to the adder. Transition **tCO_Add** consumes the token from place **pC2.** Colored petri net is used so tokens have color. In GPenSIM, color of a token is a set of strings on the token

```
pC2_tokenID = tokenAny('pC2', 1);
pC2_colors = get_color('pC2', pC2_tokenID);

msg_to_adder = [ax_square,',',bx,',',c];

client_t2 = tcpip('localhost',5000,'NetworkRole','client');
fopen(client_t2);
fwrite(client_t2,msg_to_adder)
```

Then, creates the message to be sent, request for a connection, and write the message to the network in order to send to the adder.

Final step, transition **tCI_Add** open the server and establish connection with the adder to get the arithmetic sum of ($ax^2 + bx + c$).

```
client_server1 = tcpip('0.0.0.0',9000,'NetworkRole','server');
fopen(client_server1);
received_msg_from_adder = fgetl(client_server1);


fprintf('The received sum is: %d\n\n', msg);
```

Read the network message by calling “fgetl” and display the message after received; the job is done. Now, the transition will run again to get another input, and it will do so until the values of variables all ‘0’.

4.1.2 Multiplier

Multiplier module contains five MATLAB files, Main Simulation File (MSF), Petri net Definition File (PDF), and three pre_processor files. In PDF, the transitions and places are identified and defined how they are connected. The pre_processor files have code, written the conditions how an enable transitions can fire. Now, I will explain how the multiplier module works with some code snippets.

```
%Module Multiplier PDF

function [png]=module_multiplier_pdf()
png.PN_name = 'Multiplier';
png.set_of_Ps = {'pM1', 'pM2'};
png.set_of_Ts = {'tMI', 'tM', 'tMO'};
png.set_of_As = {'tMI', 'pM1', 1, ...
                 'pM1', 'tM', 1, ...
                 'tM', 'pM2', 1, ...
                 'pM2', 'tMO', 1};
png.set_of_Ports = {'tMI','tMO'};
```

From the PDF file, we can see how the elements (transitions and places) are connected. In addition, the module has one input port (`'tMI'`) and one output port (`'tMO'`). As I mentioned before with the ports the module can communicate with other modules.

After running the multiplier module, the input transition `'tMI'` gets enable and listen to the communication channel, receives any message addressed to it.

```
multiplier_server = tcpip('0.0.0.0',7000,'NetworkRole','server');
fopen(multiplier_server);
received_packet_from_client = fgetl(multiplier_server);


msg=char([num2str(extracted_msg(1)),',',num2str(extracted_msg(2)),',
',num2str(extracted_msg(3)),',',num2str(extracted_msg(4))]);

transition.new_color={msg};
fire=1;
```

The code shows transition **tMI** acts as a server and establishes a connection when gets the connection request from the client. Read the network message by function "*fgetl*", and extract the message after receiving. Then, stores the message in place `pM1`as a token, token forms as color, and fire the transition.

After that, the local transition **tM** gets enable and do the necessary multiplication.

```
pM1_tokenID = tokenAny('pM1',1);
pM1_colors = get_color('pM1',pM1_tokenID);

%extract the values
a=str2num(char(pM1_colors(1)));
b=str2num(char(pM1_colors(2)));
c=str2num(char(pM1_colors(3)));
x=str2num(char(pM1_colors(4)));
```

Transition, **tM** consumes the token and get colors of the token from place `pM1`. Extract the message by getting individual values of the variables.

```
%Compute the values of a_x_square, b_x, and c
ax_square = a*x*x;
bx = b*x;

message =[a_xx,',', b_x,',',c_v];
transition.override = 1;
transition.new_color = {message};
fire=1;
```

After getting each values, transition calculate the values of (ax^2, *b.x, and c*) and create message. Then, save the message in place `pM2`, and fire the transition.

In last step, the output transition **tMO** gets enable and consume token from place `pM2` **.** Also, transition make the data packet of the calculated values, request TCP/IP connection, write the packet to the network by calling function *fwrite* to returns the values of (ax^2, *b.x, and c*) to the client, and fire the transitions.

```
pM2_tokenID = tokenAny('pM2', 1);
pM2_colors = get_color('pM2', pM2_tokenID);

result_back_to_client = [ax_square,',',bx,',',c];

multiplier_client = tcpip('localhost',4000,'NetworkRole','client');
fopen(multiplier_client);
fwrite(multiplier_client, result_back_to_client);
```

Finally, after firing the transition, I have opened the server (input port) globally by declaring the global variable of the server. We already know that in GPenSIM, we have a property called global visibility by which the global variables are accessible within the modules. The purpose of opening the server is to make the module active and get ready to receive the next message.

```
fire=1;


%open the server to get message from client again
multiplier_server = tcpip('0.0.0.0',7000,'NetworkRole','server');
fopen(multiplier_server);
```

We can see from the code, after transition fired I called the TCP/IP server and open it by calling *fopen* function.

4.1.3 Adder

The activities of the adder module are similar to the multiplier module. The only difference is the local transition tA performs the necessary arithmetic operations.

Adder module also contains five MATLAB files, Main Simulation File (MSF), Petri net Definition File (PDF), and three pre_processor files. In PDF, the transitions and places are identified and defined how they are connected. The pre_processor files have code, written the conditions how an enable transitions can fire.

```
%Module Adder PDF

function [png] = module_adder_pdf()
png.PN_name = 'Adder';
png.set_of_Ps = {'pA1', 'pA2'};
png.set_of_Ts = {'tAI', 'tA', 'tAO'};
png.set_of_As = {'tAI', 'pA1', 1, ...
    'pA1', 'tA', 1, ...
    'tA', 'pA2', 1, ...
    'pA2', 'tAO', 1};
png.set_of_Ports = {'tAI','tAO'};
```

PDF file represents how the elements (transitions and places) of adder module are connected. Adder module also have one input `'tAI'` and one output `'tAO'` ports same as multiplier module.

Input port `tAI` receive the message from the client and output port `tAO` send the arithmetic sum to the client, and transition **tA** only do arithmetic function.

4.2 Client-Server Model.

This model is designed to develop a distributed communication in client-server-based architecture. The system consists of three distributed Petri modules and exchange information TCP/IP sockets. Section 3.1.1 explained the overall design and purpose of these three modules, how many transitions and places are in each module, and their functions. In addition, a brief explanation was given in that section about how the whole systems work and communicate with each other. This section aims to discuss how each module is implemented, and network is developed.

4.2.1 Module A

Module A contains five MATLAB files, Main Simulation File (MSF), Petri net Definition File (PDF), and three pre_processor files. In PDF, the transitions and places are identified and defined how they are connected. The pre_processor files have code, written the conditions how an enable transitions can fire. In this section, I will explain how each transition works and how they develop a connection to communicate with other distributed modules.PDF file expresses the set of places and transitions, also shows how they are connected. This module has one input `'tAI'` and one output `'tAO'` ports, which are responsible to receive and send data over the network respectively.

```
%Module "A" pdf

function [png] = A_pdf()
png.PN_name = 'A';
png.set_of_Ps = {'pA1', 'pA2'};
png.set_of_Ts = {'tA', 'tAO', 'tAI'};
png.set_of_As = {
    'tA', 'pA1', 1, ...
    'pA1', 'tAO', 1, ...
    'tAI', 'pA2', 1};
png.set_of_Ports = {'tAI','tAO'};
```

This module generates tokens at different time and send it to module X, this module also receive messages from module X. So, local transition `'tA'` generates token at different time and save in place `'pA1'` transition `'tAI'` sends message over the network, and transition `'tAO'` receives message from the network.

When we run this module, transition 'tAI' gets enable first and run the server to check any message addressed to it. I did this by setting transition firing time.

```
global_info.TOKEN_FIRING_TIME = [10 25 40 55 70];
global_info.TOKEN_FIRING_TIME1 = [1 15 30 45 60];
```

I have set two token firing times in global variables. One is for input transition 'tAI' so that it gets active at this time and check any message in the network if so receive it, save it in place pA2, and fire the transition. Another token firing time is for transition tA, so that it can generate token at that time, and transfer the message by output transition 'tAO'.

```
time_to_generate_token = global_info.TOKEN_FIRING_TIME1(1);
ctime = current_time();

fire = 1;
```

```
if (fire==1)


%establish connection
A_server = tcpip('0.0.0.0',7000,'NetworkRole','server');
fopen(A_server);
received_packet = fgetl(A_server);


%sending acknowledgment
fwrite(A_server, 'message is received');

transition.new_color={msg};
fire = 1;
```

Therefore, the transition will generate a token at that time and fire; if the fire=1, then open the server and establish a connection. Then, read the network message by function *fgetl,* receive if there any message, send an acknowledgment message to the sender, save the received message in place pA2, and fire the transition.

Then, it will move to local transition tA. This transition only generates token by a token generator at different times and save the token in place pA1.

```
% if the variable "TOKEN_FIRING_TIME" is empty, then all
%    the firings are done; no more firing
if isempty(global_info.TOKEN_FIRING_TIME)
    fire = 0;
    return;
end


time_to_generate_token = global_info.TOKEN_FIRING_TIME(1);
ctime = current_time();

% if it is time to fire, then remoev the time from variable and fire
if ge(ctime, time_to_generate_token)
    if ge(length(global_info.TOKEN_FIRING_TIME),2)
        global_info.TOKEN_FIRING_TIME = ...
            global_info.TOKEN_FIRING_TIME(2:end);
    else
        global_info.TOKEN_FIRING_TIME = [];
    end
    transition.new_color = {'From: A', 'To: B'};
    fire = 1;
else  % it is not time to fire
    fire = 0;

 end
```

The code shows a token generator, generating token at different time, token possess two colors “From: A” and “To: B”, and save the token in place `pA1.`

Now the transition `'tAO'` consumes the token from `pA1,` and create a message with source and destination that need to be sent through TCP/IP.

```
pA1_tokenID = tokenAny('pA1',1);
pA1_colors = get_color('pA1', pA1_tokenID);


msg = [source,',',destination];
```

```
A_client = tcpip('localhost',4000,'NetworkRole','client');
fopen(A_client);
fwrite(A_client,msg);

%receiving message acknowledgment
RecAck= fgetl(A_client);
fprintf('Acknowledgment: %s\n',RecAck);
```

After creating network message, it calls a TCP/IP connection request and write the message on it. Then, wait to get acknowledgment from the receiver.

4.2.2 Module B

Module B also contains five MATLAB files, Main Simulation File (MSF), Petri net Definition File (PDF), and three pre_processor files. In PDF, the transitions and places are identified and defined how they are connected. The pre_processor files have code, written the conditions how an enable transitions can fire.

The functions of module B are the same as Module A. The only difference is first the local transition generates token and send by output transition, and then input transition gets enable and check network messages. The token generating times and transition firing conditions are different between these two modules.

```
%Module "B" pdf

function [png]=B_pdf()
png.PN_name = 'B';
png.set_of_Ps = {'pB1', 'pB2'};
png.set_of_Ts = {'tB', 'tBO', 'tBI'};
png.set_of_As = {
    'tB', 'pB1', 1, ...
    'pB1', 'tBO', 1, ...
    'tBI', 'pB2', 1};
png.set_of_Ports = {'tBI','tBO'};
```

PDF file shows the set of places and transitions also shows how they are connected. Module B also has one input 'tBI' and one output 'tBO' ports, which are responsible for receiving and sending data over the network, respectively.

```
global_info.TOKEN_FIRING_TIME = [1 15 30 45 60];
global_info.TOKEN_FIRING_TIME1 = [5 20 35 50 65];
```

Token firing times of module B.

In addition, in this module, I have used the try-catch function for the connection request. In try, every client will send a TCP/IP connection request to the server; the server might be busy with other work, or the server is not open. In that case, the connection request will move to the catch function and wait for as long the pause time is set, and then send the request again to the server.

4.2.3 Module X

This module works as transceivers, receives messages from modules A and B, and sends messages to A and B. The module contains four MATLAB files, Main Simulation File (MSF), Petri net Definition File (PDF), and two pre_processor files. In PDF, the transitions and places are identified, and defined how they are connected. The pre_processor files have code, written the conditions how an enable transitions can fire.

```
%Module X

function [png] = X_pdf()
png.PN_name = 'X';
png.set_of_Ps = {'pX0'};
png.set_of_Ts = {'tXI', 'tXO'};
png.set_of_As = {
    'tXI', 'pX0', 1, ...
    'pX0', 'tXO', 1};

png.set_of_Ports = {'tXI','tXO'};
```

PDF file represents the set of places and transitions of the module, also shows how they are connected. This module has one input `'tXI'` and one output `'tXO'` ports, which are responsible to receive and send data over the network respectively.

As it is a transceiver, first it will try to receive message and then transfer to the destination. Therefore, input port `'tXI'` opens the server and establish TCP/IP connections with the requested connection.

```
X_server = tcpip('0.0.0.0',4000,'NetworkRole','server');
fopen(X_server);

%read the network msg and receive it, either from A or B
network_msg = fgetl(X_server);


%sending acknowledgment that message is received
fwrite(X_server, 'message is received');


transition.new_color={msg};
fire = 1;
```

The code shows, transition establish a connection, read network message by calling *fgetl* function, and then sends an acknowledgment message to the sender. After that, save the received message in place `'pX0'` and fire.

Transition 'tXO' consumes token from place 'pX0', extracts message, and identify the source and destination of the message.

```
pX0_tokenID = tokenAny('pX0',1);
pX0_colors = get_color('pX0', pX0_tokenID);


%extracted message
source = pX0_colors(1);
destination = pX0 colors(2);
```

```
if strcmp(destination, 'To: A')
    disp('Sending the message to A');
    try
        %request for connection
        client_A = tcpip('localhost',7000,'NetworkRole','client');
        fopen(client_A);

        %checking connection status, open means connected
        connection_status = client_A.Status;
        fprintf('Connection status is: %s\n',connection_status);
        disp('....connected');

        %writing the message to the network
        fwrite(client_A,source)

        %receiving message acknowledgment
        RecAck= fgetl(client_A);
        fprintf('Acknowledgment: %s\n',RecAck);

        fclose(client_A);
    catch
        disp('.......server is busy!');
        disp('waiting.......');
        pause(20);

        %request for connection
        client_A = tcpip('localhost',7000,'NetworkRole','client');
        fopen(client_A);

        %checking connection status, open means connected
        connection_status = client_A.Status;
        fprintf('Connection status is: %s\n',connection_status);
        disp('....connected');

        %writing the message to the network
        fwrite(client_A,source)

        %receiving message acknowledgment
        RecAck= fgetl(client_A);
        fprintf('Acknowledgment: %s\n',RecAck);

        fclose(client A);
```

After extracting the message, it checks the source and destination of the message and connects with the destination accordingly.

From the above code, we can see that if the string in the destination is “To: A”, it will send a connection request to the server of module A. Otherwise, it will check the next statement. Else if the string in destination is “To: B” then send a connection request to the server of module B. Moreover, it will receive an acknowledgment from the receivers once the recipient receives the message.

Once the message is transferred to the intended destination, the server end of module X will be activated to receive the following message.

Chapter 5

5 Testing, Analysis and Results

This chapter presents the results from the implementation done in Chapter 4. It starts by showing the results of a distributed system for computing quadratic function in section 5.1. Then, section 5.2 shows how distributed modules exchange information in a client-server architecture. The systems were run in MATLAB; GPenSIM software was installed for simulation. For each model, three distributed modules ran in three different MATLAB instances and communicated with them. MATLAB TCP/IP sockets are used to communicate with the MATLAB instances.

5.1 Compute Quadratic Equation

This section shows the result of the quadratic equation performed by client, multiplier, and adder in distributed system.

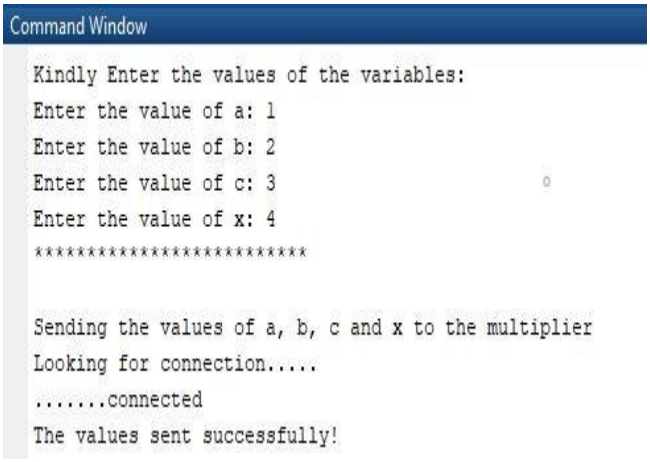

Fig. 5.1: Sending input values to multiplier

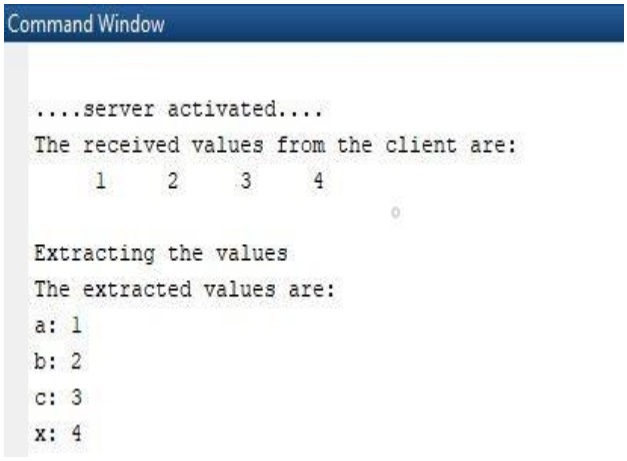

Fig. 5.2: Multiplier received input values

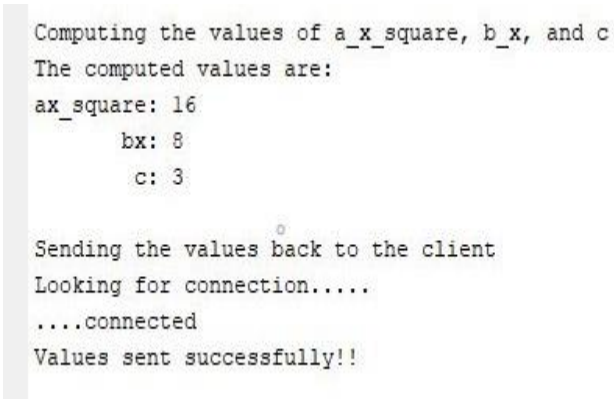

Fig. 5.3: Computed values and sending to client

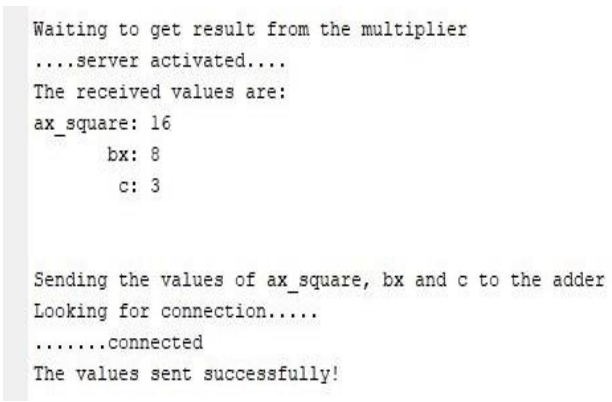

Fig. 5.4: Client received results

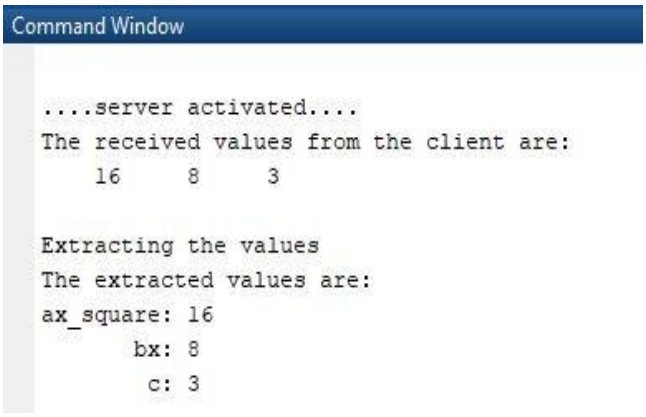

Fig. 5.5: Adder received the values

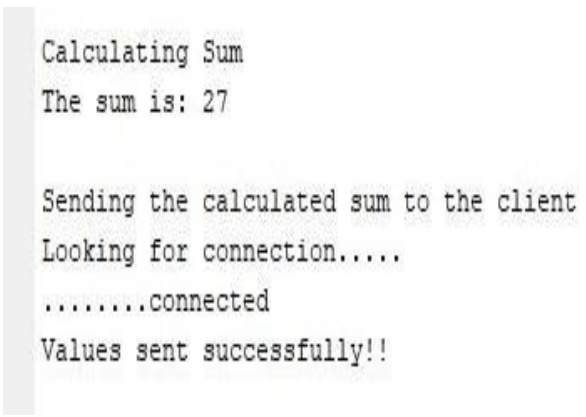

Fig. 5.6: Adder sending results to client

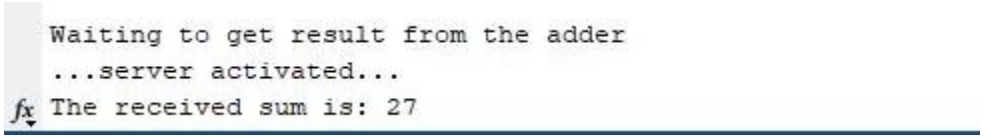

Fig. 5.7: Client received the arithmetic sum

Fig. 5.1 Shows the client getting user input for the variables and sending it to the multiplier. Fig. 5.2 express multiplier received the values from the client and extracted values. Fig. 5.3 shows calculated multiplier values of (*ax2, bx, and c*) and returns the results to the client. Fig. 5.4 presents the client received results from the multiplier and sending the values to the adder. Fig. 5.5 shows adder received the values from the client and extracted the values. Fig. 5.6 express adder done the arithmetic operations and calculated sum and returns the sum to the client. Finally, fig. 5.7 shows the client received the sum.

Fig. 5.8 represents the results after computed the quadratic function. Likewise, 5.10 and 5.11 represent the output after one job is done. Fig. 5.9 shows the client module gets stop only if the values are 0 for all the variables.

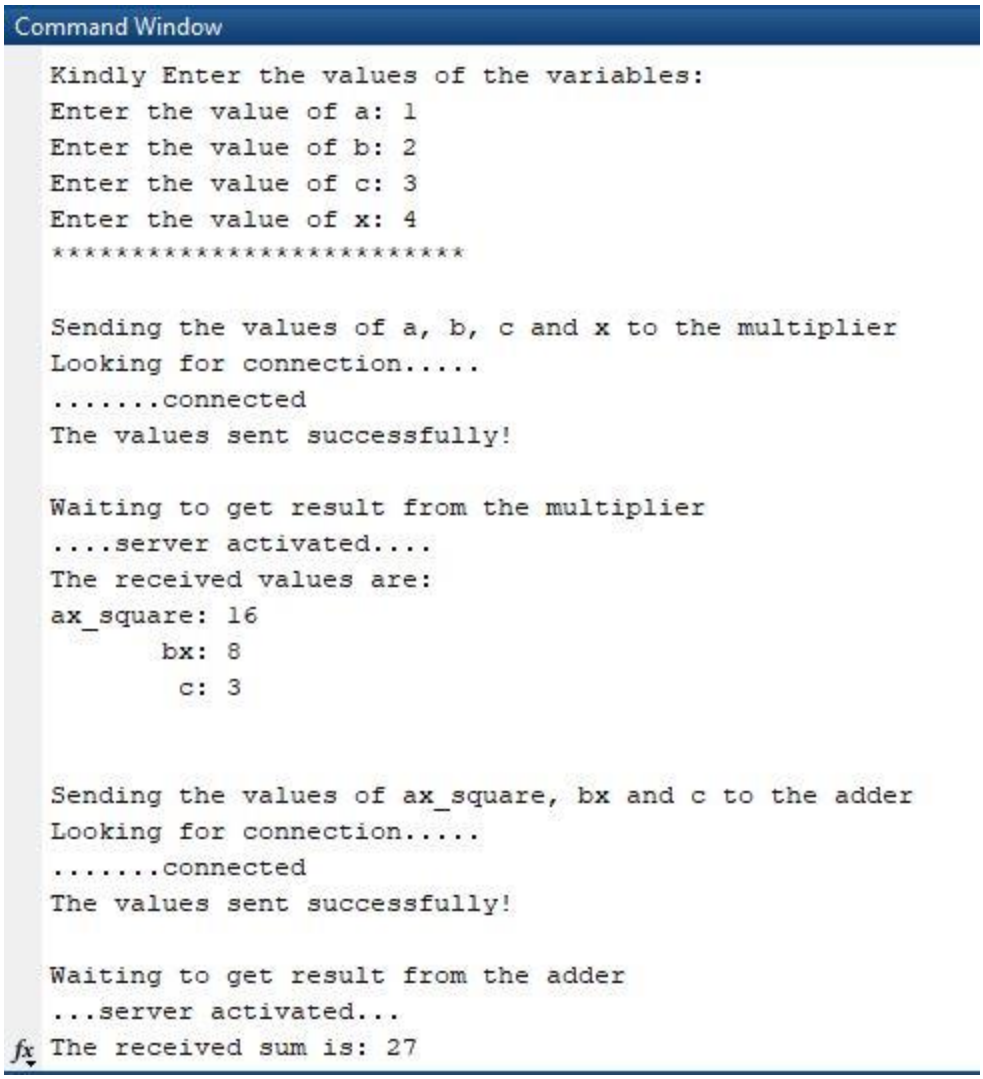

Fig. 5.8: Results of client for one iteration

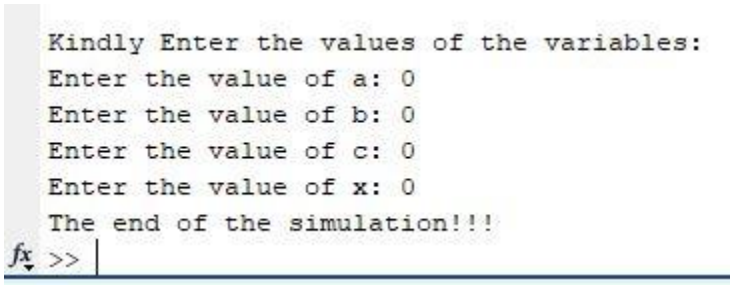

Fig. 5.9: Simulation stops if values are all zero

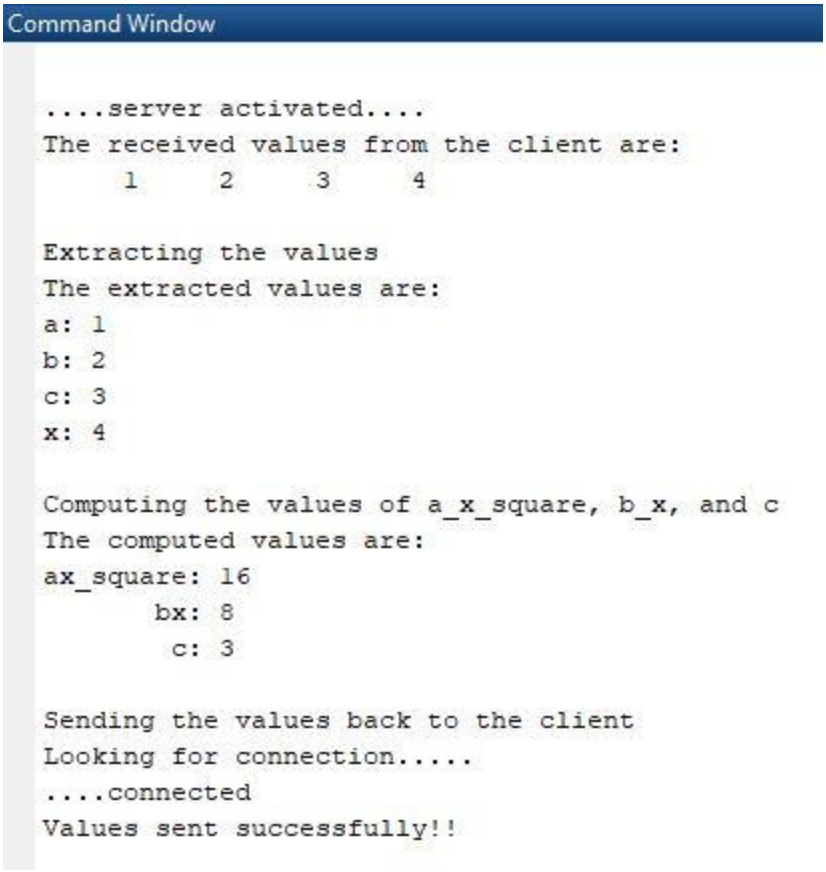

Fig. 5.10: Results of multiplier module

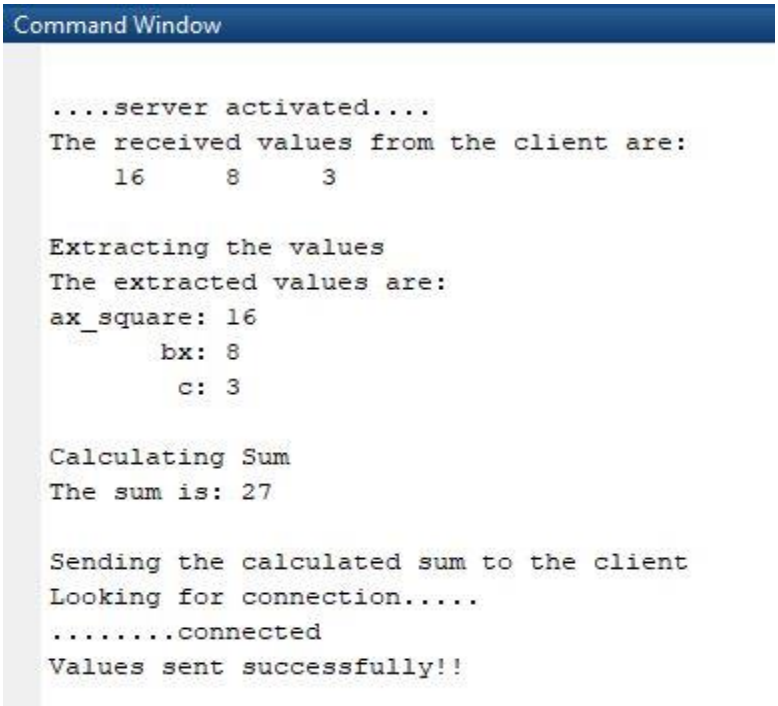

Fig. 5.11: Results of adder module

5.2 Client-Server Model

This section presents the output how the distributed modules exchange information through TCP/IP sockets in client-server based system.

Fig. 5.12: B sending message to X

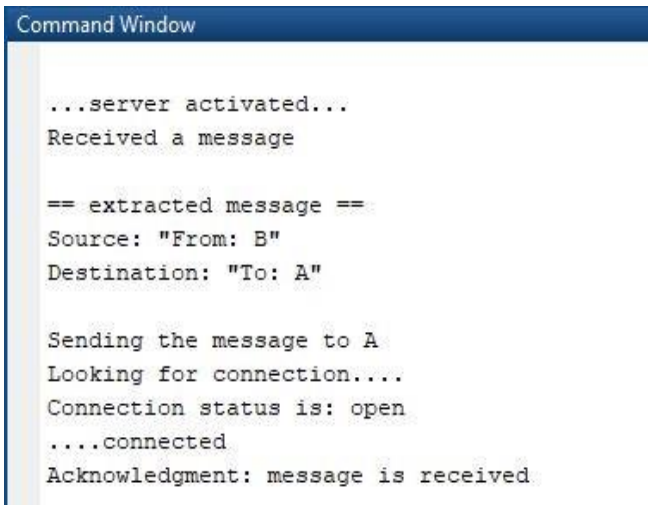

Fig. 5.13: X received message and sending to A

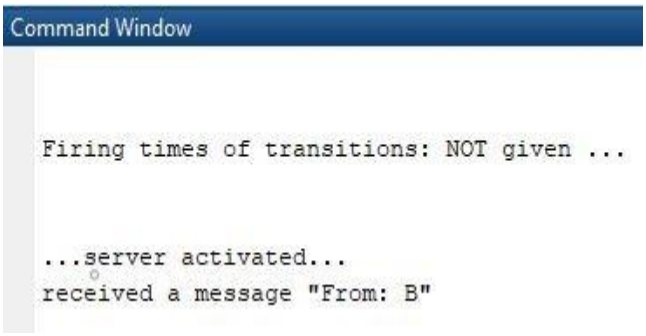

Fig. 5.14: A received message from X

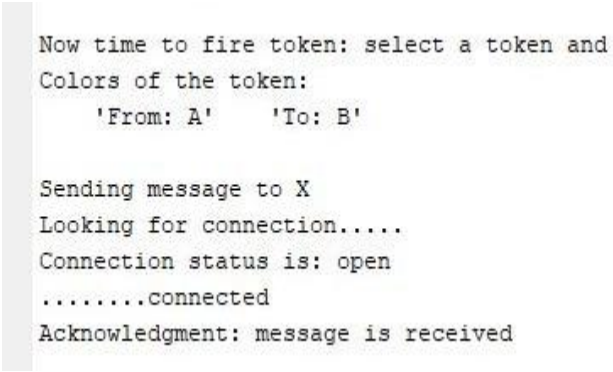

Fig. 5.15: A sending message to X

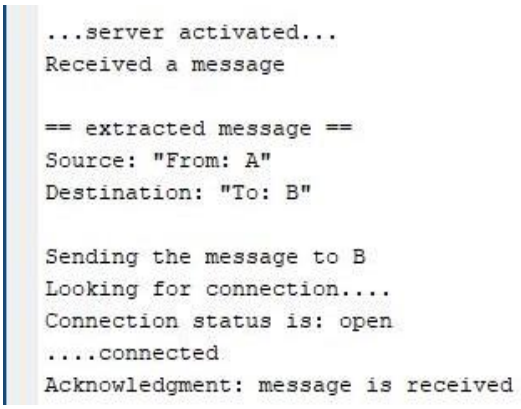

Fig. 5.16: X received message from A

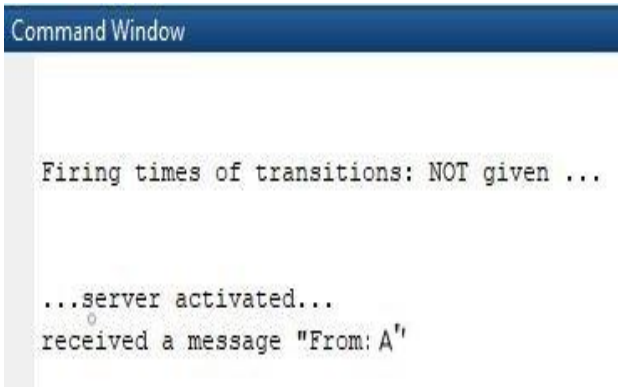

Fig. 5.17: B received message from X

Fig. 5.12 shows a token generated by module B and sending it to X; module X received the message from B in fig. 5.13, and then checked the message's source and destination, and sending the message to module A (destination). Fig. 5.14 shows that the message is received by module A. Then, module A sending a message to X in fig. 5.15. Fig.5.16 shows the message is received by X and sends an acknowledgment to the sender. Finally, the received message is transferred to module A which is shown in fig. 5.17.

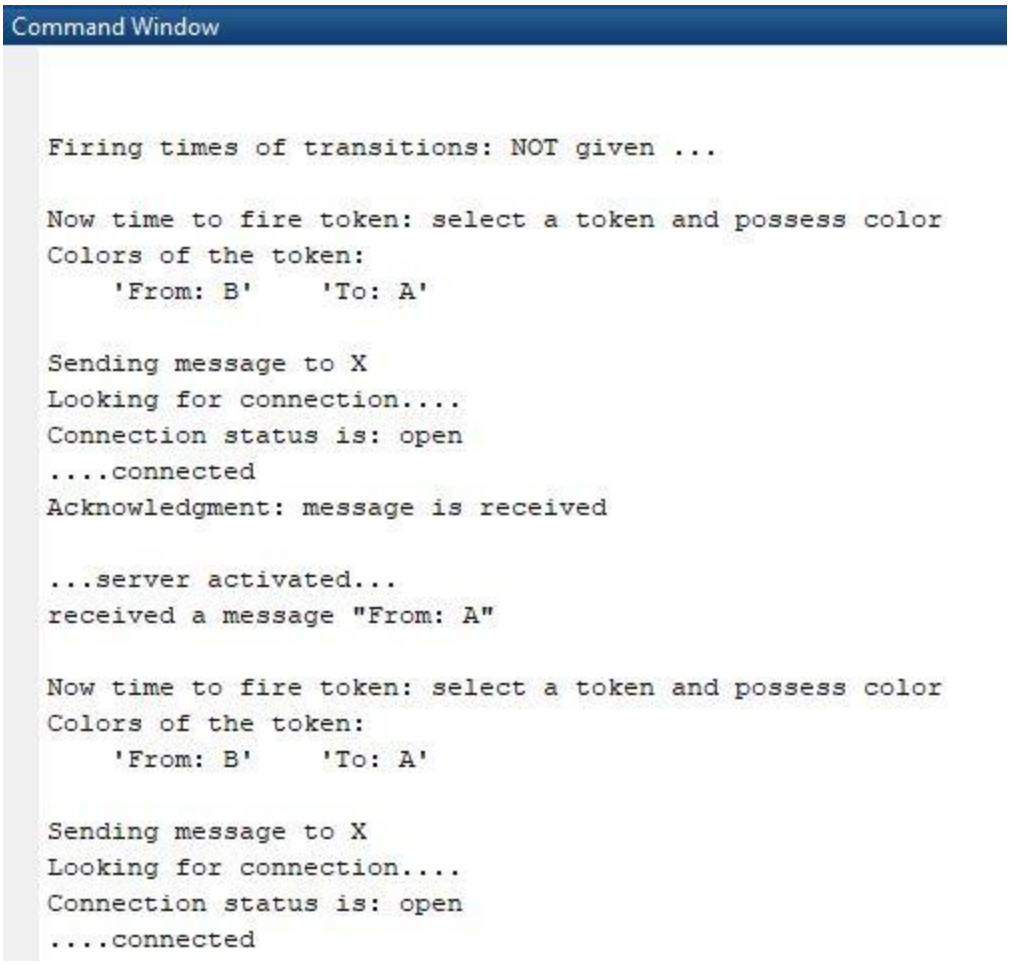

Fig. 5.18: Module B sending and receiving messages at different time

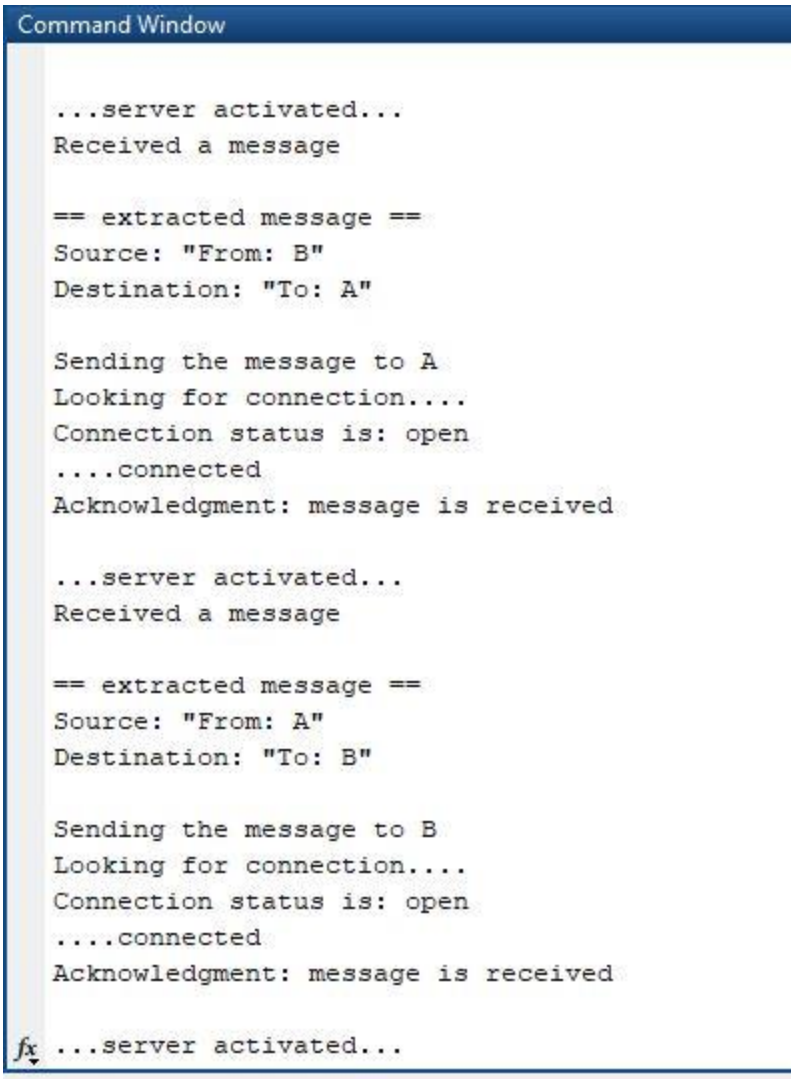

Fig. 5.19: Module X receiving and transferring messages constantly

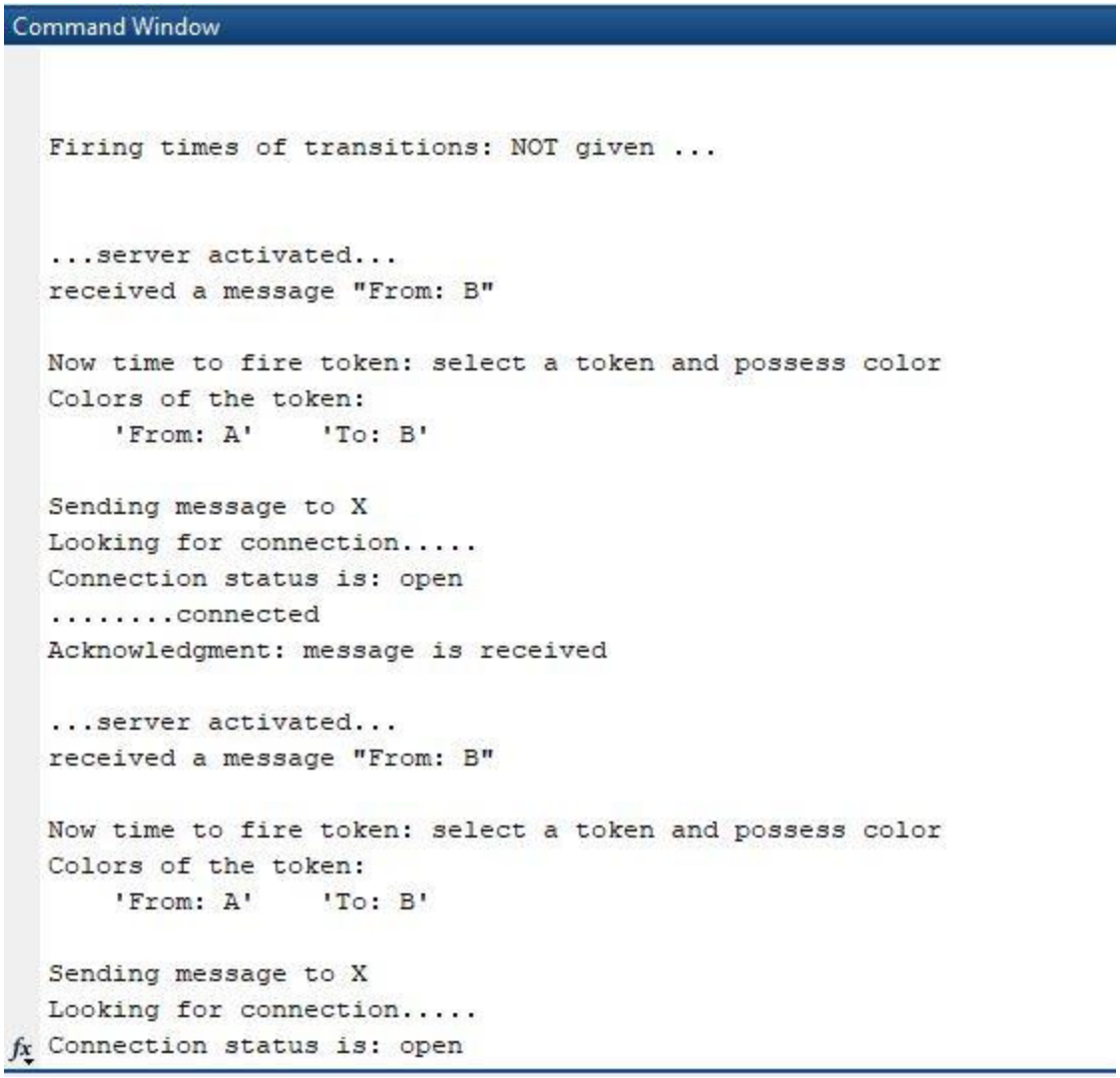

Fig. 5.20: Module A receiving and sending messages at different times

Chapter 6

6 Discussion

The main objectives of this thesis were to implement Petri Networks using the GPenSIM toolbox and integrate it with the MATLAB TCP/IP toolbox. The idea of hosting Petri modules on different computers that are geographically kept apart and the modules communicating between themselves did not exist before it was presented by Davidrajuh [11]. The network was successfully implemented, the real-time simulation results are obtained in the MATLAB command window, and the corresponding graphical results are obtained. Section 5.1 and 5.2 show how the two distributed systems work without interrupting, compute quadratic function, and exchange information in a client-server pattern. The results show that distributed Petri net modules concept will reduce the state space size and complexities for discrete event systems. Therefore, develop distributed Petri net modules with the new modular Petri net [2] and implement a network layer to connect the modules minimize the simulation time, reduce the state space size, and other complexities.

6.1 Limitations of the work

A limitation is encountered regarding the TCP/IP in Matlab. TCP/IP has some limitations in distributed communication. In this thesis, TCP/IP sockets communication has been used as a communication protocol. In MATLAB, when we open a connection either as the 'server ' or the 'client' using a 'tcpip' object, the call blocks the MATLAB execution until either the connection is made or a timeout occurred. MATLAB does not provide any provision to interrupt the tcpip socket, which makes the distributed communication a little bit complicated.

The purpose of the client-server model (3.1.2) was to develop a system where one transition (input) will always listen to the communication channel and receive any message addressed to it. Another transition will produce tokens at different times, and another transition (output) sends the token to the communication channel. My developed system is working like this, but I have generated token and set firing times for input transition as well, which can be challenging if we have a huge number of distributed modules and we want to make communication between them. Because of the limitation of the tcpip sockets, I did so. Otherwise, when this transition gets enable, it will wait to establish a connection. If there is no connection request from any client, there is no provision to interrupt it means the system cannot move to another transition because MATLAB execution is blocked. In addition, sometimes unwanted timeout occurred in tcpip means tcpip is unable to read all data.

6.2 Future work

Because so much distributed communication is focused on sending and receiving messages, message-queuing systems, similar to electronic mail systems, can be utilized as an alternative middleware service. The system's primary concept is that communication is accomplished by placing messages in queues. The message is then transmitted through several communication servers to reach its intended recipient. The message will be placed in the recipient queues, and it will be read by the recipient at some point, but there are no assurances as to when it will be read. Both the transmitter and the receiver can operate independently in this form of the communication system. When a message is transmitted to its queues, the receiver does not need to be executing. When the receiver removes a message from the queues, the sender does not need to be running.

References

[1] T. M. Sobh, "Discrete Event Dynamic Systems: An Overview," https://repository.upenn.edu/cis_reports/388, May 1991.

[2] Y. ZHENG, "Discrete Event Dynamic System:Methodology and Some Recent Progresses," Institute of Automation, Academia Sinica, Beijing, China, 1995.

[3] R. Davidrajuh, "A New Modular Petri Net for Modeling Large Discrete-Event Systems: A proposal Based on the Literature Study," MDPI, 15 November 2019.

[4] R. Davidrajuh, "GPenSIM, "General-purpose Petri Net Simulator"," http://www.davidrajuh.net/gpensim/.

[5] R. Davidrajuh, "Modular Petri Net models of Communicating Agents," ResearchGate, 2018.

[6] R. DAVIDRAJUH, "Modeling discrete-event systems with gpensim: An introduction," Springer International Publishing, 2018.

[7] V. M. X. X. SAVI, "Liveness and boundedness analysis for petri nets with event graph modules.," International Conference on Application and Theory of Petri Nets. Springer, Berlin, Heidelberg, 1992.

[8] G. G. DE JONG and B. LIN, "A communicating Petri net model for the design of concurrent asynchronous modules," 31st Design Automation Conference. IEEE, 1994.

[9] Y. XUE, R. M. KIECKHAFER and F. F. CHOOBINEH, "Automated construction of GSPN models for flexible manufacturing systems," Computers in Industry, 1998.

[10] S. Christensen and L. Petrucci, "Modular analysis of Petri nets," The computer journal, 2000..

[11] G. J. TSINARAKIS, N. C. TSOURVELOUDIS and K. P. VALAVANIS, "Modular Petri Net based modeling, analysis, synthesis and performance evaluation of random topology dedicated production systems," Journal of Intelligent Manufacturing, 2005.

[12] H. LEE and A. BANERJEE, "A modular Petri Net based architecture to model manufacturing systems exhibiting resource and timing uncertainties.," 2009 IEEE International Conference on Automation Science and Engineering, 2009.

[13] C. Mahulea, J.-M. García-Soriano and J.-M. Colom, "Modular Petri net modeling of the Spanish health system," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 2012.

[14] C. Mahulea, L. Mahulea, J. M. G. Soriano and J. M. Colom, "Modular Petri net modeling of healthcare systems," *Flexible Services and Manufacturing Journal,* vol. 30, pp. 329-357, 2018.

[15] O. BONNET-TORRÈS, " Pipe extended for two classes of monitoring Petri nets," International Conference on Application and Theory of Petri Nets. Springer, Berlin, Heidelberg, 2006.

[16] K. JENSEN and L. M. KRISTENSEN, "Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems," Communications of the ACM, 2015.

[17] S. Berger, M. Bogenreuther, B. Häckel and O. Niesel, "Modelling availability risks of IT threats in smart factory networks--a modular Petri net approach," 2019.

[18] R. DAVIDRAJUH, "Extracting Petri Modules From Large and Legacy Petri Net Models," IEEE Access, 2020.

List of Figures

List of Tables

Appendix

A1. **User Manual**

1. Install MATLAB software (2019 or updated)
2. Download and install GPenSIM.
    a. Unzip GPenSIM_v10_System_Files.zip
    b. Set MATLAB Path command
3. Download and unzip the Petri_modules.rar file
4. Select which model you want to run (Client-server or Quadratic)
5. Open a MATLAB instance for each module (3 instances for each model)
6. Add the modules to your MATLAB
7. Select the path as GPenSIM
8. To run Quadratic model
    a. Open the folder, folder contains (adder, multiplier, and client)
    b. Open three folders
    c. Open three matlab instances
    d. Run client.msf in one instance, multiplier.msf in another instance, and adder.msf in another instance.
    e. Simulation will start once you enter the input values in client end
9. To run Client-server model
    a. Open the folder, folder contains (A, B, and X)
    b. Open three different MATLAB instances
    c. First run x.msf in one instance, then A and B in two different instances.

A2. **Distributed System for Quadratic Function**

Adder

```
%Adder MSF

clear all;
clc;

global adder_server
global global_info
global_info.STOP_AT = 100000;
global network


pns = pnstruct({'module_adder_pdf'});
dyn.ft = {'allothers', 10};
pni = initialdynamics(pns, dyn);

fprintf('....server activated....\n');
adder_server = tcpip('0.0.0.0',5000,'NetworkRole','server');
fopen(adder_server);

sim = gpensim(pni);
```

```
%Module Adder PDF

function [png] = module_adder_pdf()
png.PN_name = 'Adder';
png.set_of_Ps = {'pA1', 'pA2'};
png.set_of_Ts = {'tAI', 'tA', 'tAO'};
png.set_of_As = {'tAI', 'pA1', 1, ...
    'pA1', 'tA', 1, ...
    'tA', 'pA2', 1, ...
    'pA2', 'tAO', 1};
png.set_of_Ports = {'tAI','tAO'};
```

```
%tAI_pre
function [fire,transition]=tAI_pre(transition)
global adder_server
global network
global global_info


%read the network message
received_packet = fgetl(adder_server);


% We will display the message once we will have some information
% we will ignore the empty message
if ~isempty(received_packet)
    %extrcting the information from the received packet
    extracted_msg=received_packet;

    %converting the information to number
    extracted_msg = str2num(char(extracted_msg));
    disp('The received values from the client are: ');
    disp(extracted_msg)


    %constructing message and converting to char
    msg =
char([num2str(extracted_msg(1)),',',num2str(extracted_msg(2)),',',num2str(ext
racted_msg(3))]);

    transition.new_color={msg};
    fire=1;
else
    fire=0;
end
```

```
%tA_pre
function [fire,transition]=tA_pre(transition)
global network
global global_info


%select token from 'pA1'
pA1_tokenID = tokenAny('pA1',1);

%getting colors
pA1_colors = get_color('pA1',pA1_tokenID);


%splitting the stream to get independent values of ax_square, bx and c by
%considering ',' as the delimiter
pA1_colors=strjoin(pA1_colors);
pA1_colors=char(pA1_colors);
pA1_colors=string(pA1_colors);
```

```
pA1_colors=split(pA1_colors,',');


disp('Extracting the values');
%converting the received strings caharcter into number for the further
%computation and processing
ax_square = str2num(pA1_colors(1));
bx = str2num(pA1_colors(2));
c = str2num(pA1_colors(3));

disp('The extracted values are:');
fprintf('ax_square: %d\n', ax_square);
fprintf('       bx: %d\n', bx);
fprintf('        c: %d\n\n', c);


disp('Calculating Sum')
%computing sum
sum = ax_square+bx+c;
adder_sum=num2str(sum);
fprintf('The sum is: %d\n\n',sum);

transition.override=1;
transition.new_color={adder_sum};
fire=1;

end
```

```
%tAO_pre
function [fire,transition]=tAO_pre(transition)
global network
global global_info
global adder_server


%select token from 'pA2'
pA2_tokenID = tokenAny('pA2',1);

%getting colors
pA2_colors=get_color('pA2',pA2_tokenID);


%converting the values from string to number
pA2_colors = char(pA2_colors);
pA2_colors = str2num(pA2_colors);

sum = num2str(pA2_colors);
```

```
%making connection with client to send the computed sum
disp('Sending the calculated sum to the client');
disp('Looking for connection.....');
adder_client = tcpip('localhost',9000,'NetworkRole','client');
fopen(adder_client);
disp('........connected');

result_to_client = sum;
fwrite(adder_client,result_to_client)
disp('Values sent successfully!!');

fclose(adder_client);
fire=1;

%open the server to get message from client again
adder_server = tcpip('0.0.0.0',5000,'NetworkRole','server');
fopen(adder_server);
fprintf('\n....server activated....\n');

end
```

Multiplier

```
% Multiplier MSF

clear all;
clc;

global multiplier_server
global global_info
global_info.STOP_AT = 100000;
global network


pns = pnstruct('module_multiplier_pdf');
dyn.ft = {'allothers',10};
pni = initialdynamics(pns, dyn);

fprintf('....server activated....\n');
multiplier_server = tcpip('0.0.0.0',7000,'NetworkRole','server');
fopen(multiplier_server);

sim = gpensim(pni);
```

```
%Module Multiplier PDF

function [png]=module_multiplier_pdf()
png.PN_name = 'Multiplier';
png.set_of_Ps = {'pM1', 'pM2'};
png.set_of_Ts = {'tMI', 'tM', 'tMO'};
png.set_of_As = {'tMI', 'pM1', 1, ...
                 'pM1', 'tM', 1, ...
                 'tM', 'pM2', 1, ...
                 'pM2', 'tMO', 1};
png.set_of_Ports = {'tMI','tMO'};
```

```
%tMI_pre
function [fire,transition]=tMI_pre(transition)
global multiplier_server
global network
global global_info


%read the network message
received_packet_from_client = fgetl(multiplier_server);

%We will display the message once we will have some information and
%we will ignore the empty message
if ~isempty(received_packet_from_client)
    %extrcting the information from the received packet
    extracted_msg = received_packet_from_client;

    %converting the information to number
    extracted_msg = str2num(char(extracted_msg));
    disp('The received values from the client are: ');
    disp(extracted_msg)

    %constructing message and converting to char

msg=char([num2str(extracted_msg(1)),',',num2str(extracted_msg(2)),',',num2str
(extracted_msg(3)),',',num2str(extracted_msg(4))]);

    transition.new_color={msg};
    fire=1;
else
    fire=0;
end
```

```
%tM_pre
function [fire,transition]=tM_pre(transition)


%select token from 'pM1'
pM1_tokenID = tokenAny('pM1',1);

%getting colors
pM1_colors = get_color('pM1',pM1_tokenID);


%splitting the stream to get independent values of a, b and c by
%considering ',' as the delimiter
pM1_colors = strjoin(pM1_colors);
pM1_colors = char(pM1_colors);
pM1_colors = string(pM1_colors);
pM1_colors = split(pM1_colors,',');


disp('Extracting the values');
%converting the received strings caharcter into number for the further
%computation and processing
a=str2num(char(pM1_colors(1)));
b=str2num(char(pM1_colors(2)));
c=str2num(char(pM1_colors(3)));
x=str2num(char(pM1_colors(4)));

disp('The extracted values are:');
fprintf('a: %d\n', a);
fprintf('b: %d\n', b);
fprintf('c: %d\n', c);
fprintf('x: %d\n\n', x);


disp('Computing the values of a_x_square, b_x, and c');
%ax_square
ax_square = a*x*x;
a_xx = num2str(ax_square);

%bx
bx = b*x;
b_x = num2str(bx);

%c
c_v = num2str(c);


disp('The computed values are: ');
fprintf('ax_square: %d\n', ax_square);
fprintf('       bx: %d\n', bx);
fprintf('        c: %d\n\n', c);

%making a token using comma as delimeter and save to place for further
process
```

```
message =[a_xx,',', b_x,',',c_v];


transition.override = 1;
transition.new_color = {message};
fire=1;

end
```

```
%tMO_pre
function [fire,transition]=tMO_pre(transition)
global network
global global_info
global multiplier_server

%select token from 'pM2'
pM2_tokenID = tokenAny('pM2', 1);

%getting colors
pM2_colors = get_color('pM2', pM2_tokenID);


%splitting the stream to get independent values of ax_square, bx and c by
%considering ',' as the delimiter
pM2_colors = split(pM2_colors,',');
pM2_colors = char(pM2_colors);
pM2_colors = str2num(pM2_colors);


%extracting the value of ax_square, bx and c, and converting them from
%number to string
ax_square = num2str(pM2_colors(1));
bx = num2str(pM2_colors(2));
c = num2str(pM2_colors(3));


disp('Sending the values back to the client')
disp('Looking for connection.....');
multiplier_client = tcpip('localhost',4000,'NetworkRole','client');
fopen(multiplier_client);
disp('....connected');

result_back_to_client = [ax_square,',',bx,',',c];
fwrite(multiplier_client, result_back_to_client);
disp('Values sent successfully!!');

fclose(multiplier_client);
fire=1;


%open the server to get message from client again
multiplier_server = tcpip('0.0.0.0',7000,'NetworkRole','server');
```

```
fopen(multiplier_server);
fprintf('\n....server activated....\n');


end
```

Client

```
% Client MSF

clear all;
clc;
global global_info
global_info.STOP_AT = 100000;
global network


pns = pnstruct('module_client_pdf');
dyn.m0={'pC0', 1};
dyn.ft={'allothers',10};
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

disp('The end of the simulation!!!');
```

```
%Module Client PDF

function [png] = module_client_pdf()
png.PN_name = 'Client';
png.set_of_Ps = {'pC0', 'pC1', 'pC2', 'pC3'};
png.set_of_Ts = {'tCO_Mul', 'tCI_Mul', ...
                 'tCO_Add', 'tCI_Add'};
png.set_of_As = {
                 'pC0','tCO_Mul',1, ...
                 'tCO_Mul','pC1',1, ...
                 'pC1','tCI_Mul',1, ...
                 'tCI_Mul','pC2',1, ...
                 'pC2','tCO_Add',1, ...
                 'tCO_Add','pC3',1, ...
                 'pC3','tCI_Add',1, ...
                 'tCI_Add','pC0',1};
png.set_of_Ports = {'tCO_Mul', 'tCI_Mul', 'tCO_Add', 'tCI_Add'};
```

```
%tCO_Mul
function [fire,transition]=tCO_Mul_pre(transition)
global global_info
global client_mul


disp('Kindly Enter the values of the variables: ')
a = input('Enter the value of a: ');
b = input('Enter the value of b: ');
c = input('Enter the value of c: ');
x = input('Enter the value of x: ');

if ((a == 0) && (b == 0) && (c == 0) && (x == 0))
    global_info.STOP_SIMULATION = 1;
else
    %constructing the client message

client_message=char([num2str(a),',',num2str(b),',',num2str(c),',',num2str(x)]
);

    disp('**************************');
    fprintf('\nSending the values of a, b, c and x to the multiplier\n');

    %making connection to the multiplier
    disp('Looking for connection.....');
    client_mul = tcpip('localhost',7000,'NetworkRole','client');
    fopen(client_mul);
    disp('.......connected');

    %sending the values
    fwrite(client_mul,client_message);
    fprintf('The values sent successfully!\n\n');
    fclose(client_mul);

end
fire=1;

end
```

```
%tCI_Mul
function [fire,transition]=tCI_Mul_pre(transition)
global global_info
global client_server2


%open the server to get result values from the multiplier
disp('Waiting to get result from the multiplier');
client_server2 = tcpip('0.0.0.0',4000,'NetworkRole','server');
fopen(client_server2);
disp('....server activated....');

%receiving the values sent by multiplier
read_msg_from_mult = fgetl(client_server2);
```

```
if ~isempty(read_msg_from_mult)
    received_msg = str2num(char(read_msg_from_mult));

    %displaying the values
    disp('The received values are:');
    fprintf('ax_square: %d\n',received_msg(1));
    fprintf('       bx: %d\n',received_msg(2));
    fprintf('        c: %d\n',received_msg(3));


    msg =
[num2str(received_msg(1)),',',num2str(received_msg(2)),',',num2str(received_m
sg(3))];
    transition.override = 1;
    transition.new_color={msg};

end
    fire = 1;
end
```

```
%tCO_Add
function [fire,transition]=tCO_Add_pre(transition)
global global_info
global client_t2


%select token from 'pM2'
pC2_tokenID = tokenAny('pC2', 1);

%getting colors
pC2_colors = get_color('pC2', pC2_tokenID);

%splitting the stream to get independent values of ax_square, bx and c by
%considering ',' as the delimiter
pC2_colors = split(pC2_colors,',');
pC2_colors = char(pC2_colors);
pC2_colors = str2num(pC2_colors);


%extracting the value of ax_square, bx and c, and converting them from
%number to string
ax_square = num2str(pC2_colors(1));
bx = num2str(pC2_colors(2));
c = num2str(pC2_colors(3));


%making connection with adder
fprintf('\n\nSending the values of ax_square, bx and c to the adder\n');
disp('Looking for connection.....');
client_t2 = tcpip('localhost',5000,'NetworkRole','client');
fopen(client_t2);
disp('.......connected');
```

```
%sending extracted values to adder
msg_to_adder = [ax_square,',',bx,',',c];
fwrite(client_t2,msg_to_adder)
fprintf('The values sent successfully!\n\n');

fclose(client_t2);
fire = 1;

end
```

```
%tCI_Add
function [fire,transition]=tCI_Add_pre(transition)
global global_info
global client_server1


disp('Waiting to get result from the adder');
client_server1 = tcpip('0.0.0.0',9000,'NetworkRole','server');
fopen(client_server1);
fprintf('...server activated...\n');
received_msg_from_adder = fgetl(client_server1);

if ~isempty(received_msg_from_adder)
    msg = received_msg_from_adder;
    msg = str2num(char(msg));

    fprintf('The received sum is: %d\n\n', msg);

end
    fclose(client_server1);
    fire = 1;
end
```

A3. **Client-server model**

Module A

```
%Module A MSF

clear all;
clc;
global global_info

global_info.TOKEN_FIRING_TIME = [10 25 40 55 70];
global_info.TOKEN_FIRING_TIME1 = [1 15 30 45 60];
global_info.STOP_AT = 100;
global_info.DELTA_TIME = 1;
```

```
pns = pnstruct('A_pdf');

%dyn.m0 = {'pA1', 0}; % tokens initially
%dyn.ft = {'tA', 1, 'allothers', 10};
%pni = initialdynamics(pns, dyn);

pni = initialdynamics(pns);

sim = gpensim(pni);
prnss(sim);
```

```
%Module "A" pdf

function [png] = A_pdf()
png.PN_name = 'A';
png.set_of_Ps = {'pA1', 'pA2'};
png.set_of_Ts = {'tA', 'tAO', 'tAI'};
png.set_of_As = {
    'tA', 'pA1', 1, ...
    'pA1', 'tAO', 1, ...
    'tAI', 'pA2', 1};
png.set_of_Ports = {'tAI','tAO'};
```

```
%tAI_pre
function [fire,transition]=tAI_pre(transition)
global A_server
global network
global global_info



% if the variable "TOKEN_FIRING_TIME" is empty, then all
%    the firings are done; no more firing
if isempty(global_info.TOKEN_FIRING_TIME1),
    fire = 0;
    return;
end

time_to_generate_token = global_info.TOKEN_FIRING_TIME1(1);
ctime = current_time();

% if it is time to fire, then remoev the time from variable and fire
if ge(ctime, time_to_generate_token),
    if ge(length(global_info.TOKEN_FIRING_TIME1),2),
        global_info.TOKEN_FIRING_TIME1 = ...
            global_info.TOKEN_FIRING_TIME1(2:end);
    else
        global_info.TOKEN_FIRING_TIME1 = [];
    end
    fire = 1;
else  % it is not time to fire
    fire = 0;
```

```
end

if (fire==1)


%open the server to receive message
A_server = tcpip('0.0.0.0',7000,'NetworkRole','server');
fopen(A_server);
fprintf('\n...server activated...\n');

%read the network message
received_packet = fgetl(A_server);

%We will display the message once we will have some information and
%we will ignore the empty message
if ~isempty(received_packet)
    fprintf('received a message "%s"\n\n', received_packet);

    %sending acknowledgment
    fwrite(A_server, 'message is received');

    %converting to character
    msg=char(received_packet);

    transition.new_color={msg};
    fire = 1;
end

end
```

```
%tA_pre
function [fire,transition]=tA_pre(transition)
global network
global global_info



% if the variable "TOKEN_FIRING_TIME" is empty, then all
%    the firings are done; no more firing
if isempty(global_info.TOKEN_FIRING_TIME)
    fire = 0;
    return;
end


time_to_generate_token = global_info.TOKEN_FIRING_TIME(1);
ctime = current_time();

% if it is time to fire, then remoev the time from variable and fire
if ge(ctime, time_to_generate_token)
```

```
    if ge(length(global_info.TOKEN_FIRING_TIME),2)
        global_info.TOKEN_FIRING_TIME = ...
            global_info.TOKEN_FIRING_TIME(2:end);
    else
        global_info.TOKEN_FIRING_TIME = [];
    end
    transition.new_color = {'From: A', 'To: B'};
    fire = 1;
else  % it is not time to fire
    fire = 0;

 end
```

```
%tAO_pre
function [fire,transition]=tAO_pre(transition)
global network
global global_info
global A_client



disp('Now time to fire token: select a token and possess color');

%select a token from 'pA1'
pA1_tokenID = tokenAny('pA1',1);

%getting colors
disp('Colors of the token: ');
pA1_colors = get_color('pA1', pA1_tokenID);
disp(pA1_colors);


source = char(string(pA1_colors(1)));
destination = char(string(pA1_colors(2)));


msg = [source,',',destination];


% making connection with X server to send the message
disp('Sending message to X');
disp('Looking for connection.....');

try
    %request for connection
    A_client = tcpip('localhost',4000,'NetworkRole','client');
    fopen(A_client);

    %checking connection status, open means connected
    connection_status = A_client.Status;
    fprintf('Connection status is: %s\n',connection_status);
```

```
    disp('........connected');

    %writing the message to the network
    fwrite(A_client,msg);

    %receiving message acknowledgment
    RecAck= fgetl(A_client);
    fprintf('Acknowledgment: %s\n',RecAck);

    fire=1;
    fclose(A_client);
catch
    disp('......waiting');
    disp('Server is busy.....');
    pause(20);

    %request for connection
    A_client = tcpip('localhost',4000,'NetworkRole','client');
    fopen(A_client);

    %checking connection status, open means connected
    connection_status = A_client.Status;
    fprintf('Connection status is: %s\n',connection_status);
    disp('........connected');

    %writing the message to the network
    fwrite(A_client,msg);

    %receiving message acknowledgment
    RecAck= fgetl(A_client);
    fprintf('Acknowledgment: %s\n',RecAck);

    fire=1;
    fclose(A_client);
end


end
```

Module B

```
% Module "B" MSF

clear all;
clc;
global global_info

global_info.TOKEN_FIRING_TIME = [1 15 30 45 60];
global_info.TOKEN_FIRING_TIME1 = [5 20 35 50 65];
global_info.STOP_AT = 100;
global_info.DELTA_TIME = 1;
```

```
pns = pnstruct('B_pdf');

%dyn.m0 = {'pB1', 0}; % tokens initially
%dyn.ft = {'tB', 1, 'allothers', 10};
%pni = initialdynamics(pns, dyn);

pni = initialdynamics(pns);

sim = gpensim(pni);
prnss(sim);
```

```
%Module "B" pdf

function [png]=B_pdf()
png.PN_name = 'B';
png.set_of_Ps = {'pB1', 'pB2'};
png.set_of_Ts = {'tB', 'tBO', 'tBI'};
png.set_of_As = {
    'tB', 'pB1', 1, ...
    'pB1', 'tBO', 1, ...
    'tBI', 'pB2', 1};
png.set_of_Ports = {'tBI','tBO'};
```

```
%tBI_pre
function [fire,transition]=tBI_pre(transition)
global B_server
global network
global global_info


% if the variable "TOKEN_FIRING_TIME" is empty, then all
%    the firings are done; no more firing
if isempty(global_info.TOKEN_FIRING_TIME1),
    fire = 0;
    return;
end

time_to_generate_token = global_info.TOKEN_FIRING_TIME1(1);
ctime = current_time();

% if it is time to fire, then remoev the time from variable and fire
if ge(ctime, time_to_generate_token)
    if ge(length(global_info.TOKEN_FIRING_TIME1),2)
        global_info.TOKEN_FIRING_TIME1 = ...
            global_info.TOKEN_FIRING_TIME1(2:end);
    else
        global_info.TOKEN_FIRING_TIME1 = [];
    end
    fire = 1;
else  % it is not time to fire
```

```
    fire = 0;

end

if (fire==1)


%open the server to receive message
B_server = tcpip('0.0.0.0',3000,'NetworkRole','server');
fopen(B_server);
fprintf('\n...server activated...\n');

%read the network message
received_packet = fgetl(B_server);

%We will display the message once we will have some information and
%we will ignore the empty message
if ~isempty(received_packet)
    fprintf('received a message "%s"\n\n', received_packet);

    %sending acknowledgment
    fwrite(B_server, 'message is received');

    %converting to character
    msg=char(received_packet);

    transition.new_color={msg};
    fire = 1;

end

end
```

```
%tB_pre
function [fire,transition]=tB_pre(transition)
global network
global global_info



% if the variable "TOKEN_FIRING_TIME" is empty, then all
%    the firings are done; no more firing
if isempty(global_info.TOKEN_FIRING_TIME)
    fire = 0;
    return;
end


time_to_generate_token = global_info.TOKEN_FIRING_TIME(1);
ctime = current_time();
```

```
% if it is time to fire, then remoev the time from variable and fire
if ge(ctime, time_to_generate_token)
    if ge(length(global_info.TOKEN_FIRING_TIME),2)
        global_info.TOKEN_FIRING_TIME = ...
            global_info.TOKEN_FIRING_TIME(2:end);
    else
        global_info.TOKEN_FIRING_TIME = [];
    end
    transition.new_color = {'From: B', 'To: A'};
    fire = 1;
else  % it is not time to fire
    fire = 0;

 end
```

```
%tBO_pre
function [fire,transition]=tBO_pre(transition)
global network
global global_info
global B_client



disp('Now time to fire token: select a token and possess color');

%select a token from 'pB1'
pB1_tokenID = tokenAny('pB1', 1);

%getting colors
disp('Colors of the token: ');
pB1_colors = get_color('pB1', pB1_tokenID);
disp(pB1_colors);


source = char(string(pB1_colors(1)));
destination = char(string(pB1_colors(2)));

msg = [source,',',destination];


% making connection with X server to send the message
disp('Sending message to X');
disp('Looking for connection....');

try
    %request for connection
    B_client = tcpip('localhost',4000,'NetworkRole','client');
    fopen(B_client);

    %checking connection status, open means connected
```

```
    connection_status = B_client.Status;
    fprintf('Connection status is: %s\n',connection_status);
    disp('....connected');

    %writing the message to the network
    fwrite(B_client, msg);

    %receiving message acknowledgment
    RecAck= fgetl(B_client);
    fprintf('Acknowledgment: %s\n',RecAck);

    fire=1;
    fclose(B_client);
catch
    disp('.......server is busy!');
    disp('waiting.......');
    pause(30);

    %request for connection
    B_client = tcpip('localhost',4000,'NetworkRole','client');
    fopen(B_client);

    %checking connection status, open means connected
    connection_status = B_client.Status;
    fprintf('Connection status is: %s\n',connection_status);
    disp('....connected');

    %writing the message to the network
    fwrite(B_client, msg);

    %receiving message acknowledgment
    RecAck= fgetl(B_client);
    fprintf('Acknowledgment: %s\n',RecAck);

    fire=1;
    fclose(B_client);
end


end
```

Module X

```
% Module X MSF

clear all;
clc;
global X_server
global global_info
global network
```

```
pns = pnstruct('X_pdf');
dyn.ft={'allothers',10};
pni = initialdynamics(pns, dyn);

disp('...server activated...');
X_server = tcpip('0.0.0.0',4000,'NetworkRole','server');
fopen(X_server);

sim = gpensim(pni);
```

```
%Module X

function [png] = X_pdf()
png.PN_name = 'X';
png.set_of_Ps = {'pX0'};
png.set_of_Ts = {'tXI', 'tXO'};
png.set_of_As = {
    'tXI', 'pX0', 1, ...
    'pX0', 'tXO', 1};

png.set_of_Ports = {'tXI','tXO'};
```

```
%tXI_pre
function [fire,transition]=tXI_pre(transition)
global X_server
global global_info
global network


%read the network msg and receive it, either from A or B
network_msg = fgetl(X_server);

if ~isempty(network_msg)
    disp('Received a message');

    %sending acknowledgment that message is received
    fwrite(X_server, 'message is received');

    %conveerting message to char
    msg = char(network_msg);

    transition.new_color={msg};
    fire = 1;
else
    fire = 0;

end

end
```

```
%tX0_pre
function [fire,transition]=tX0_pre(transition)
global client_A
global X_server
global client_B
global global_info
global network


%select a token from 'pX0'
pX0_tokenID = tokenAny('pX0',1);

%getting colors
pX0_colors = get_color('pX0', pX0_tokenID);

%extracted message
pX0_colors = split(pX0_colors,',');

source = pX0_colors(1);
destination = pX0_colors(2);


source = char(strjoin(source));
destination = char(strjoin(destination));

fprintf('\n== extracted message ==\n');
fprintf('Source: "%s"\n', source);
fprintf('Destination: "%s"\n\n', destination);


%making connection to send messages
if strcmp(destination, 'To: A')
    disp('Sending the message to A');
    disp('Looking for connection....');
    try
        %request for connection
        client_A = tcpip('localhost',7000,'NetworkRole','client');
        fopen(client_A);

        %checking connection status, open means connected
        connection_status = client_A.Status;
        fprintf('Connection status is: %s\n',connection_status);
        disp('....connected');

        %writing the message to the network
        fwrite(client_A,source)

        %receiving message acknowledgment
        RecAck= fgetl(client_A);
        fprintf('Acknowledgment: %s\n',RecAck);
```

```
        fclose(client_A);
    catch
        disp('.......server is busy!');
        disp('waiting.......');
        pause(20);

        %request for connection
        client_A = tcpip('localhost',7000,'NetworkRole','client');
        fopen(client_A);

        %checking connection status, open means connected
        connection_status = client_A.Status;
        fprintf('Connection status is: %s\n',connection_status);
        disp('....connected');

        %writing the message to the network
        fwrite(client_A,source)

        %receiving message acknowledgment
        RecAck= fgetl(client_A);
        fprintf('Acknowledgment: %s\n',RecAck);

        fclose(client_A);
    end

elseif strcmp(destination, 'To: B')
    disp('Sending the message to B');
    disp('Looking for connection....');
    try
        %request for connection
        client_B = tcpip('localhost',3000,'NetworkRole','client');
        fopen(client_B);

        %checking connection status, open means connected
        connection_status = client_B.Status;
        fprintf('Connection status is: %s\n',connection_status);
        disp('....connected');

        %writing the message to the network
        fwrite(client_B,source)

        %receiving message acknowledgment
        RecAck= fgetl(client_B);
        fprintf('Acknowledgment: %s\n',RecAck);

        fclose(client_B);
    catch
        disp('.......server is busy!');
        disp('waiting.......');
        pause(20);

        %request for connection
        client_B = tcpip('localhost',3000,'NetworkRole','client');
```

```
        fopen(client_B);

        %checking connection status, open means connected
        connection_status = client_B.Status;
        fprintf('Connection status is: %s\n',connection_status);
        disp('....connected');

        %writing the message to the network
        fwrite(client_B,source)

        %receiving message acknowledgment
        RecAck= fgetl(client_B);
        fprintf('Acknowledgment: %s\n',RecAck);

        fclose(client_B);
    end
else
    disp('No messages in the channel');

end

fire = 1;

%open the server to receive messges from clients
X_server = tcpip('0.0.0.0',4000,'NetworkRole','server');
fopen(X_server);
fprintf('\n...server activated...\n');

end
```