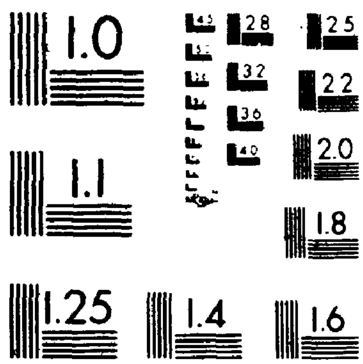
1



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

THE HARMONY OPERATING SYSTEM DESCRIBED BY PETRI NETS

by

YAO LI, B.ENG. (EE)

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering

Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario

ISBN 0-315-33431-2

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis,

"The Harmony Operating System Described by Petri Nets"

submitted by YAO LI, B.ENG.(EE), in partial fulfillment of the requirements for the degree of Master of Engineering.

F. Hadr

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

September 1986

Abstract

This thesis addresses the modeling of the Harmony operating system by Petri nets. With the preliminary descriptions of the algorithms used in Harmony, the Petri nets models are built up for system initialization, interrupt handling, message passing, task creation and destruction, error handling, and common aspects of server implementation by placing modeling emphasis on synchronization and concurrency. A new concept called double numbered token introduced to Petri nets is applied to message passing. Together with the descriptions of algorithms, the Petri nets models alleviate the unsatisfied situation—lack of documentation for Harmony. Some improvements to the source code are suggested during modeling. The deadlock detection and prevention in message passing are intensively studied as well.

Acknowledgements

My sincere thanks go to members of the ARTT Project: Prof. F. Hadziomerovic my supervisor, Prof. C.M. Woodside the director of the ARTT Project, Mr. K. Rowe the ARTT Project manager and Mr. V. Wan my colleague, for their support, guidance and discussions.

I would like to express many thanks to Prof. D.D. Falconer for his help.

The financial support from the ARTT Project is gratefully appreciated.

I dedicate this thesis to my parents.

Table of Contents

PART	I	INTRODUCTION	1
Chapter	1	General Introduction	2
	1.1	Motivation	2
	1.2	Summary of Contents	3
	1.3	My Original Contributions	4
Chapter	2	Introduction to Petri Nets	6
	2.1	Petri Nets as a Modeling Tool	6
	2.2	Petri Nets as a Performance Analysis Tool	7
Chapter	3	Introduction to Harmony	11
	3.1	Outline of Harmony	11
	3.2	Harmony Kernel	13
PART	II	MODELING OF HARMONY	16
Chapter	4	System Initialization	17
	4.1	Introduction to Algorithm	17
	4.2	Petri Nets Models	20
Chapter	5	Interrupt Handling	25
	5.1	Direct Interrupt Mode	26
	5.2	Transparent Interrupt Mode	32
Chapter	6	Message Passing	36
	6.1	Description of the Algorithms	36

6.2 High Level Petri Nets Models 43

6.3 Low Level Petri Nets Models 49

6.3.1 _Reply(rply, id) 51

6.3.2 _Try_receive(rqst, id) 51

6.3.3 _Receive(rqst, id) 55

6.3.4 _Send(rqst, rply, id) 58

6.3.5 General Models 60

6.3.6 The Double Colored Token PN Models 60

6.4 Deadlock and its Prevention 66

Chapter 7 Task Creation and Destruction 72

7.1 Introduction 72

7.2 High Level Petri Nets Models 78

7.2.1 Task Creation 79

7.2.2 Task Destruction 81

7.3 Low Level Petri Nets Models 86

7.3.1 Task Creation 86

7.3.2 Task Destruction 92

Chapter 8 Error Handling 99

8.1 Description of Algorithm 99

8.2 Petri Nets Model 101

Chapter 9 Kernel Supported Server Implementation 106

9.1 Decomposition Description 106

9.1.1 Implementing Servers 106

9.1.2 Implementing Connections 108

9.1.3 Stream I/O 108

9.1.4 Monitoring Connections 109

9.1.5 System Task 109

9.1.6 Server Tasks 110

9.2 Algorithm and Dependency Descriptions 110

9.2.1 Server Creation, Initialization and Registration 111

9.2.2 Monitoring Connections 116

9.2.3 Open a Connection 117

9.2.4 Using Connections (Stream I/O) 119

9.2.5 Close a Connection 119

9.2.6 General Dependency Description 121

9.3 Detailed Petri Nets Models 121

9.3.1 Server Creation, Initialization and Registration 121

9.3.1.1 One Initialization Record 121

9.3.1.2 Several Initialization Records 128

9.3.2 Open and Close a Connection 135

9.3.2.1 Open a Connection 135

9.3.2.2 Close a Connection 138

PART III DISCUSSIONS 141

Chapter 10 Conclusion and Future Work 142

10.1 Conclusion 142

10.2 Future Work 144

Bibliography 147

Appendix A Index of Harmony Functions Used 150

Appendix B Modified "case UNQ_RECEIVER" in _Td_service Primitive 155

Appendix C C Code of Deadlock Prevention Mechanism in Message Passing 157

Appendix D Index of Depicted Data Structures and Organizations 168

PART I

INTRODUCTION

Chapter 1 General Introduction

A model of a computer operating system is preferable when measurement and evaluation is needed. The Harmony operating system described by Petri nets is such a model that shows the parallelism and synchronization of operations. In this chapter, we present the motivation of whole research work, give a brief summary of work being done, and identify the original contribution being made.

1.1 Motivation

Modeling is an approach to study a system. It allows us to concentrate on the important features of the system under study. A model actually is a representation of the system with emphasis on the interested features.

Petri net is an effective tool for modeling, analysis of model's properties, and performance analysis. It is simple to understand, powerful in modeling synchronized and concurrent phenomena, easy to do some nets analysis, and performance evaluation.

Harmony is a real time operating system of reasonable complexity, and is difficult to comprehend. We hope that modeling by Petri nets will help in understanding of Harmony: revealing concepts involved, suggesting improvements and enabling performance evaluation.

So far, two attempts have been made to model operating systems by Petri nets. Noe [14] modeled the CDC 6400 in 1971. Because he used an ad hoc Petri nets, generality is consequently lost. Best [1] modeled the SOLO in

1976, but his work is not available. Wan [22] modeled Harmony using Finite State Machine technique. His work is very helpful in understanding of Harmony, though it neither covers whole kernel, nor goes down to very low level.

1.2 Summary of Contents

The entire Harmony, excluding various servers' implementations, is modeled by Petri nets (PN). The modeling includes system initialization, interrupt handling, message passing, task creation and destruction, error handling, and common aspects of server implementation. For easier understanding, a description of the algorithm, together with data structures and calling graphs are usually given in front of the PN model. If the topic under study is rather complex, the PN model is given in high and low levels. High level model provides an overall view, whereas low level model provides necessary details.

There is either direct or conceptual correspondence between two levels. Direct correspondence means that a transition in a high level model can be expanded to a subnet in the low level [14]. Conceptual correspondence means that, unlike the above, the correspondence can only be found conceptually.

During modeling, the emphasis was given to synchronization and concurrency. Sequential operations and uninterested subroutines are merged into transitions as much as possible. The above consideration has an impact on how deep the model should go down. Then, it is up to a researcher to include level of details.

PN models can be used for performance evaluations. One way of doing it is to submit PN models as an input to the GSPNA (generalized stochastic Petri nets automatic) software package [9], which maps PN into Markov chains and then calculates steady state probabilities. Since the GSPNA does not accept inhibitor arcs, used for zero testing of places in our models, their equivalents need to be developed.

1.3 My Original Contributions

I have made contributions to both PNs and Harmony. As to PNs, the pseudo ordinary Petri nets are applied to the modeling of a complex operating system. The pseudo ordinary Petri nets here means that when doing performance analysis the inhibitor arc can be replaced by a properly defined random switch, and the colored net can be changed to the ordinary net without losing the correctness. Compared to Noe's work [14], the PNs used here allow system modeling in more details.

To correctly model the general case of multiple tasks communicating with each other in message passing, a concept called double numbered token is suggested, that requires the modifications of the firing rules.

It is explicitly pointed out in last chapter that the GSPN can include inhibitor arcs, because the inhibitor arcs are reducible.

As the contributions to Harmony, the PN model is an abstraction of synchronization and concurrency inherent in the source code, and is an interpretation of the code implementation. So it serves as a Harmony documentation.

Also some questions are raised and improvements are suggested for the code implementation. Deadlock in message passing is studied and the mechanism of preventing it is suggested with the code implementation in C language.

Chapter 2 Introduction to Petri Nets

The original idea of Petri nets was developed by C.A. Petri in his Ph.D. dissertation in 1966 [17]. Since then the Petri nets were extensively studied mainly as a modeling tool. In 1981, Peterson gave a clear and detailed summary of Petri nets in the milestone work [16]. From 1976, researchers directed their efforts in turning PNs to a performance analysis tool. In this chapter, we briefly introduce the two aspects of Petri nets: as a modeling tool and as a performance analysis tool.

2.1 Petri Nets as a Modeling Tool

Petri nets are a tool for the study of the systems. A Petri net comprises four parts: a set of places P, a set of transitions T, a set of directed arcs A, and an initial marking M^0:

$$PN = (P, T, A, M^0)$$

$$P = \{p_1, p_2, ..., p_n\}$$

$$T = \{t_1, t_2, ..., t_m\}$$

$$A \subset \{P \times T\} \cup \{T \times P\}$$

$$M^0 = \{m_1^{\,0}, m_2^{\,0}, ..., m_n^{\,0}\}$$

A place (drawn in a circle) is an input to a transition if an arc exists from the place to the transition. A place is an output from a transition if an arc exists from the transition to the place. A marking is an assignment of tokens to the places of a Petri net. A transition is enabled when all of its input places contain at least one token. A transition may fire if it is enabled (classical definition). A transition takes zero time to fire by removing one token from each its input place and putting one token in each output place. Multiple tokens are absorbed from multiple input arcs and produced for multiple output arcs. Each firing of a transition produces a new marking. A place is k-bound or k-safe if the number of tokens in that place cannot exceed an integer k. The reachability set is defined as the set of all markings that can be reached from the initial marking M^0 by means of a sequence of transition firings.

Petri nets have been found a great deal of use in modeling various systems, such as computer software/hardware, queuing networks, physical systems, social systems, etc. Petri nets are especially suitable for modeling the systems with synchronization and concurrency; concurrency is modeled in a natural and convenient way.

After Petri nets model has been built up, one can analyze it so hopefully gain some profound understanding of the original system, thus possibly improve the system. Several techniques have been developed for the analysis of the Petri nets. Two major ones are using reachability tree and matrix equations. They provide the solutions for some of following questions: safeness, boundedness, conservation, liveness, reachability, coverability and firing sequences. Of course, the solution is preferably implemented on computer. The details of this part can be found in Peterson's book [16].

2.2 Petri Nets as a Performance Analysis Tool

In last decade, researchers have done a lot of work on turning PN to a performance analysis tool. Time, as a critical parameter, is introduced to PN via a variety of ways. The resulted PN may be called Timed Petri Nets (TPN).

Time can be either assigned to the place [2] or to the transition. For the second case, time itself may be either fixed or random. For fixed time, there can be one type of time, such as either a fixed firing rate [7] or a fixed length of firing time [18], [24]; or two types of times, for instance, either a minimum and a maximum execution time [11], or an enabling time and a firing time [19]. When random time is assigned to transitions, PN comes to SPN [13] and GSPN [10] of interest.

The SPN (stochastic Petri nets) proposed by Molloy [13] is defined by assigning an exponentially distributed firing rate to each transition in a PN for continuous time systems or a geometrically distributed firing rate for discrete time systems. A formal definition of a SPN is the following:

$$SPN = (P, T, A, M^0, R)$$

where $R = \{r_1, r_2, \ldots, r_m\}$ is the set of firing rates associated with transitions.

Molloy has shown that the SPN is isomorphic to homogeneous Markov process due to the memoryless property of the exponential distribution of firing times [12]. SPN markings correspond to MC states. In particular, k-bounded Petri nets are isomorphic to finite Markov processes.

SPN is a bridge between PN models and MC models. It allows analysts to model the (computer) systems easily with powerful PN, and translate it into MC model by using a procedure to automatically generate the reachability set of the underlying Petri net, that is, the steady states of MC, then solve it to get performance measures.

A disadvantage of SPN comes from indistinguishably assigning a nonzero time to each transition. That could lead to the state (reachable markings) explosion as system size and complexity increase, because all markings are tangible states (enabling the timed transitions only) in which the process spends nonzero time and must be counted for solving MC, whereas in GSPN the process spends zero time in vanishing states (enabling at least one immediate transition) and those states are not taken account for solving MC. Moreover, often for practical modeling purpose, it is not desirable to assign a random time to each transition at all times. So Marson generalized SPN to GSPN [10].

There are two classes of transitions: timed and immediate transitions in GSPN. Timed transitions fire after a random, exponentially distributed enabling time. Immediate transitions fire once they are enabled. Immediate transition can be considered as an extreme case of timed transition when the enabling time goes to zero. GSPN is still equivalent to the MC.

Several transitions may be enabled at the same time. If they are all timed transitions, then each fires with probability

$$Pr(t_i) = \frac{r_i}{\sum_{k=1}^{N} r_k}$$

where N is the number of timed transitions in this group. If there is one immediate transition among several enabled timed transitions, then it fires always. If all enabled transitions are immediate ones, then a probability distribution (switching distribution) is needed to select the firing transition. The switching distribution may be marking dependent. If the probability associated with an enabled immediate transition is zero, the transition cannot fire.

The inhibitor arcs, used for zero testing of a place, increase modeling power [16], and simplify graphical representation. In fact, the inhibitor arc and the random switch (a set of enabled immediate transition and the associated probability distribution) are mutually reducible. We can also replace the inhibitor arc with two classes of transitions. Examples are given in Chapter 10.

Chapter 3 Introduction to Harmony

Harmony is a multitasking, multiprocessing operating system for real-time control [3]. It is mainly written in C programming language with small amount of assembler for the ease of porting. In the following sections, we outline Harmony with its main features, and introduce its kernel. References for this chapter come from [3], [4], [5], [6], [15], [20] and [22]. Detailed descriptions of the Harmony algorithms are moved to PART II.

3.1 Outline of Harmony

Harmony is ported to a tightly coupled multiprocessor system. Among its realizations, one is on MC68010 available here. It makes four assumptions about hardware.

— Linear addressing within the system for all processors: this means that a unique address is assigned to each memory location hence they are addressable by any processor in the system.

— Direct interrupt capability: each processor supports multiple interrupt levels.

— Transparent interrupt capability: a processor can interrupt any other processors and itself.

— A test-and-set operation: it provides mutual exclusion for globally accessible data structure.

Harmony is designed for real-time control. Therefore, programs interact with the real world through I/O devices, and time is considered as a critical resource [3, p3] and [6, p5]. So in Harmony, the timeslicing and round-robin scheduling policies, as used in conventional operating systems, are replaced by the priority scheduling. Each processor supports such a scheduling system. Each task has a fixed priority level throughout its life time. A FIFO ready queue exists for each priority level. The first task on the highest priority nonempty queue is executing on the processor, and continues to execute until blocked or preempted by a higher priority task.

Multiprocessing, as another feature, means that a multiprocessor system is used in Harmony; the number of processors can be increased easily without modifying the application software, like system tasks, servers, etc. Multiprocessing is supported by the inter-processor interrupt which provides intertask communication.

Multitasking is also a Harmony feature. Tasks are tied to the specific processor. Tasks can be dynamically created and destroyed. Once a task created, it competes for using resources independently in the same way its creator does. Communication and synchronization between tasks is realized through the message passing. Message passing in Harmony is implemented by four primitives: _Send(), _Receive(), _Try_receive() and _Reply().

The next worth-mentioned feature is that Harmony is an open system. That implies that Harmony can run on many different processors and peripherals. Servers, in Harmony, are referred to as managers of both logical and physical resources. Open further implies that these servers can be easily added, deleted or modified in terms of requirements.

3.2 Harmony Kernel

Harmony consists of a kernel and a collection of tasks, and it can be split into three layers:

1. Outer (application) layer: both system tasks and user tasks reside in this layer. Four system tasks are:

   _Directory() : provides task id from its symbolic name.

   _Gossip() : provides a general reporting and logging mechanism for other tasks.

   _Local_task_manager() : responsible for task creation and destruction.

   _Idle_task() : a dummy task to absorb processor time when all other tasks are blocked.

   The first two tasks reside on processor 0 only, whereas the other two reside on all processors.

2. Middle (interface) layer: Harmony primitives reside in this layer. It can be viewed as an interface between application layer and the lowest layer. In terms of their functions, these primitives can be respectively grouped into memory management, task creation and destruction, message passing, interrupt, error handling, using connection, stream I/O, and implementing servers.

3. Inner layer (scheduler): provides task scheduling and synchronization service. Functions in this layer are mostly written in assembler to increase the execution speed. We list them below:

   _Block() : removes the caller from its ready queue which is owned by the caller's processor, and dispatches the next ready task. Returns when the blocked task becomes ready.

_Signal_processor(id) : notifies the processor of task requiring service. The id of the task specified by the parameter is written in the mailbox of the processor which executes that task and an interrupt is generated to that processor.

_Block_signal_processor(id) : removes the caller from its ready queue. The id of the caller (not specified by the parameter) is written in the mailbox of the processor that executes the task specified by the parameter. An interrupt is generated to that processor. Then the next ready task is dispatched.

_Disable() : disables all interrupts by changing the current processor priority to interrupt level 6.

_Enable() : enables interrupts by changing the current processor priority to the active task—the caller's priority.

_Idle_loop() : code for _Idle_task. It stops the processor and waits for the next interrupt.

_IP_int() : the second level inter-processor interrupt handler. Fetches id of a task requiring service and calls _Td_service() to serve the request.

_Setup() : initialization code executed by all processors but processor 0 before executing Harmony program.

_Setup0() : initialization code executed by processor 0 before executing

Harmony program.

In the above classification, the kernel is made up of the middle and inner layers. And the kernel is distributed on each processor, each with its own set of data structures. Since the lengthy part of Harmony is the variety of servers, it is not easy to locate them in our layer scheme. We shall put server tasks in the outer layer, the second level interrupt handlers written in assembler in the inner layer, and put the other functions in the middle layer.

PART II

MODELING OF HARMONY

Chapter 4 System Initialization

System initialization naturally comes first. The initialization consists of establishing each processor's C environment: building and initializing its Harmony and having the first task dispatched [3]. In this chapter, we first describe the initialization work by illustrating the calling graph and a key data structure called multiprocessor gates. Secondly, we detail the initialization procedure by using a Petri net model.

4.1 Introduction to Algorithm

Initialization is mainly done in either routine _Setup0 for processor 0, or _Setup for any other processor. They both establish the C environment, call Harmony kernel to build and initialize Harmony then dispatch a task.

Establishing the C environment for a processor involves setting up an idle task and the exception vector table, as well as clearing its mailbox slot. Next, a call to routine _I_harmony initializes its Harmony operating system. Calling graph is given in Figure 4.1, where rectangle denotes a function/routine, rhombus a task, arrow a function/routine call and dashed arrow creating a task. The depth of study is only down to the function of each routine/task involved. There is not much synchronization within each routine, but the understanding of manipulation of some data structures is required. Called routines and created tasks are listed below:

_I_extern() : performs initialization of global variables.

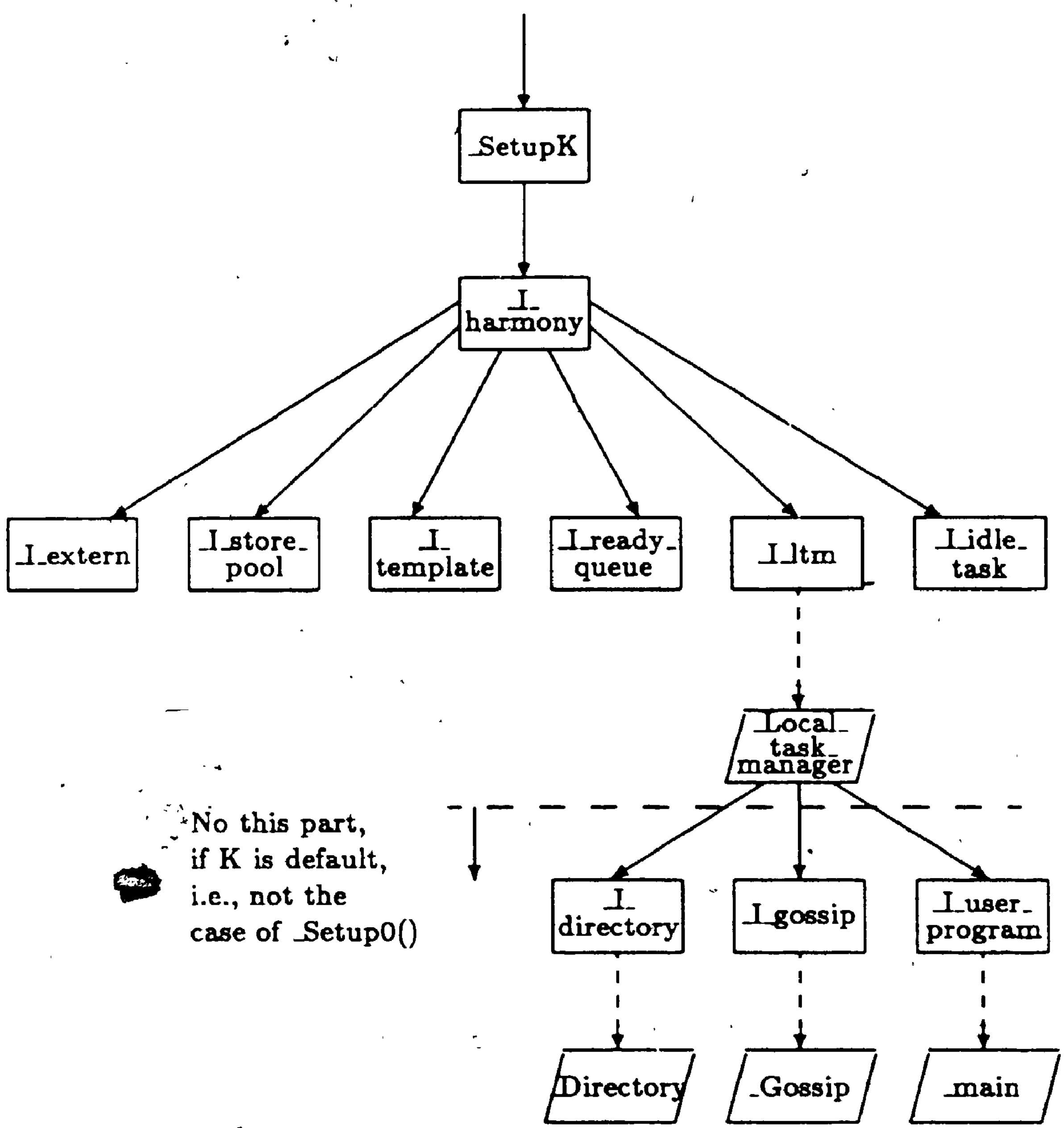


Figure 4.1. Calling Graph of System Initialization

_I_store_pool() : initializes storage pools.

_I_templates() : initializes the task templates.

_I_ready_queues() : initializes all ready queues which are at different levels of priorities.

_I_ltm() : creates the local task manager.

_I_idle_task() : creates the idle task.

_Local_task_manager() : is the root function, which is the code that the task will execute when it's dispatched, for the local task manager task. It looks after the task creation and destruction by waiting for messages and then performing the requests.

_I_directory() : creates the directory task for task template 2.

_I_gossip() : creates the gossip task for task template 3.

_I_user_program() : creates the first user task for task template 1.

_Directory() : is the root function for the directory task, which executes on processor 0 only.--It provides information for opening connections between server and client. A client submits the symbolic name of a server to _Directory(), and _Directory provides the server task's id for the client.

_Gossip() : is the root function for the gossip task which executes on processor 0. It reports any errors occurred to the user.

_main() : is the root function for the first user task. It does whatever the user wants it to do.

Note that only in the case of _Setup0, the _Local_task_manager creates three tasks: _Directory, _Gossip and _main. They reside on processor 0.

Finally, each processor other than processor 0 dispatches an idle task, while processor 0 dispatches the first user task.

A data structure called _MP_gate, multiprocessor gate, is designed to implement the synchronization. The number of gates is equal to the number of processors minus one (except processor 0). Each gate is used to control and indicate the progress of the initialization of its corresponding processor. The snapshots of the _MP_gate are given in Figure 4.2, together with explanations.

4.2 Petri Nets Model

Some synchronization is required during the system initialization. This is modeled by the PN in Figure 4.3 and corresponding Table 4. Initially, there is one token in each P_{1-i}, which represents an available processor. All together there are N + 1 processors, where N = MAX_PROC (maximum processor number). There are no tokens in any other places. Places P_{6-} represent the multiprocessor gates. The number of tokens in them denotes their gate values. For example, three tokens present denote value 3.

Initially, P_{6-} are reset to 0s. Immediate transitions t_{2-} are enabled and fire. Processors other than processor 0 enter their busywait loops (t_{2-}, P_{5-}, t_{3-} and P_{1-}) with the delay modeled by enabling timed transitions t_{3-}. After back to P_{1-}, for a specific processor, if its gate is still closed, ie., t_{4-i} is disabled, it fires t_{2-i} again.

Initially, timed transition t_1, which includes subroutine calls, is enabled for processor 0. At t_1, processor 0 executes some codes to accept download of

MP_gate	Explanations	Who sets gate value
1: 0 2: 0 3: 0 ⋮ *MAX_PROC*: 0	Multiprocessor gates are closed. Processors busywait for gates open to start their initializations.	By processor K, where K $\neq 0$
1: 1 2: 1 3: 1 ⋮ *MAX_PROC*: 1	Multiprocessor gates are open. Processor 0 has built and initialized its Harmony environment. Other processors are allowed to start their initializations.	By processor 0
1: 1 2: 2 3: 2 ⋮ *MAX_PROC*: 1	Initializations of processors, whose gate bytes marked 2, are in progress.	By processor K, where K $\neq 0$
1: 3 2: 2 3: 3 ⋮ *MAX_PROC*: 2	Processors, whose gate bytes marked 3, have completed their initializations.	By processor K, where K $\neq 0$

Figure 4.2. Snapshots of Multiprocessor Gates

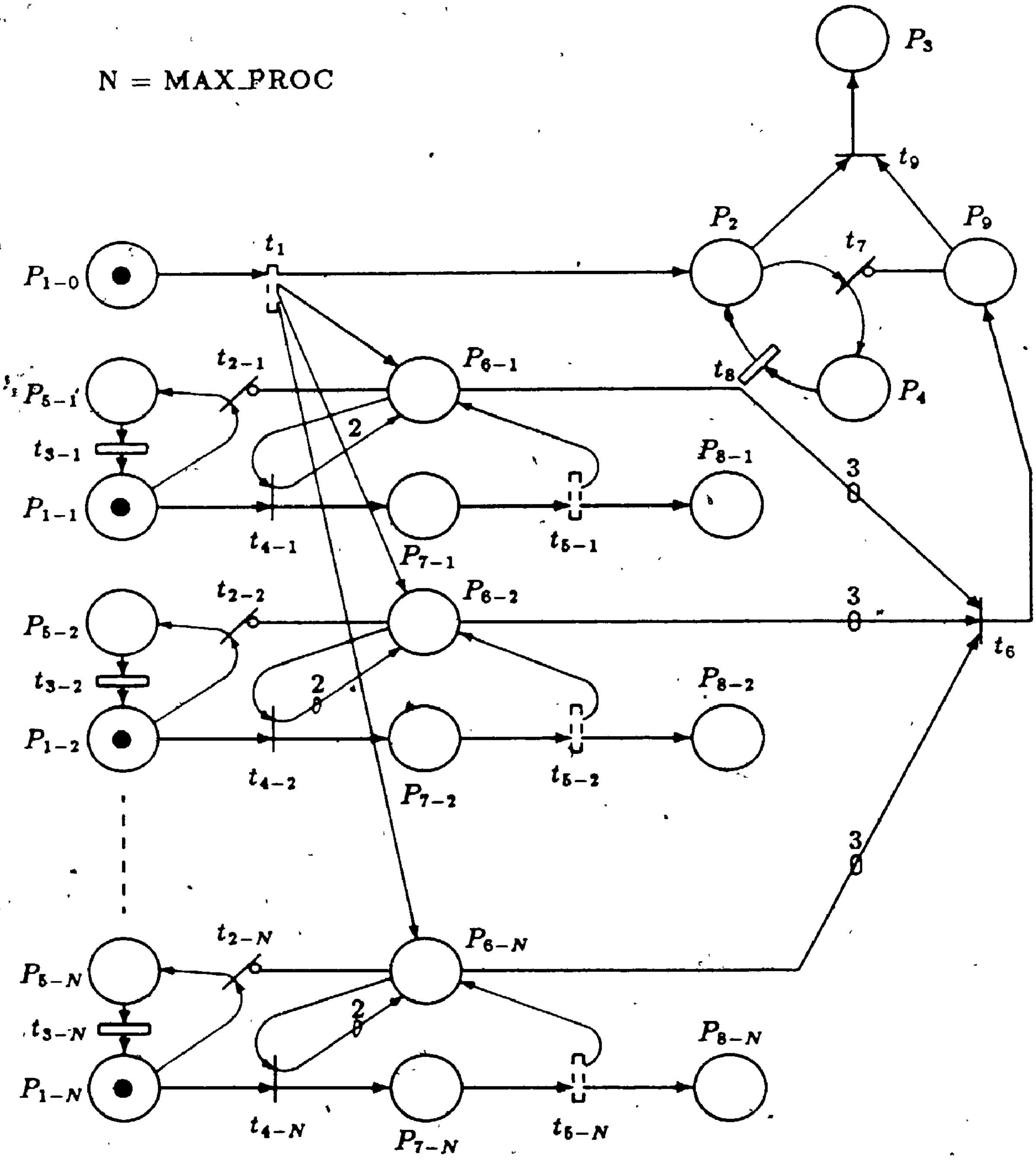


Figure 4.3 Synchronization During System Initialization

Table 4 System Initialization

Transition	Meaning
t_1	sets up C environment, builds and initializes its Harmony kernel, opens gates (executed on processor 0)
t_{2-}	enter the busywait loop for other processors
t_{3-}	delay in the busywait loop
t_{4-}	mark initializations in progress, set up their C environments
t_{5-}	build and initialize other processors' Harmony kernels, mark initializations completed in _MP_gate, dispatch idle tasks
t_6	informs processor 0 of all other processors having completed their initializations, lets processor 0 go
t_7	processor 0 enters the wait loop
t_8	causes delay in wait loop for processor 0
t_9	dispatches the first user task
Place	**Meaning**
P_{1-}	hold idle processors
P_2	processor 0 waits in the wait loop for other processors to complete their initializations
P_3	processor 0 is executing the first user task
P_4	transit place in the wait loop for processor 0
P_{5-}	transit places in busywait loops for other processors
P_{6-}	multiprocessor gates
P_{7-}	initialization of processor k is in progress
P_{8-}	processors are executing their idle tasks
P_9	for a control token to release processor 0

executable codes. The end of downloading causes it to activate _Setup0, which initializes its kernel, and opens the gates (puts one token each to P_{6-i}).

After gates open, t_{4-i} fires, which changes the gate value to 2 (takes one token from P_{6-i}, returns two to it), indicating its initialization is in progress. Then t_{5-i} fires, executes _Setup on/for i-th processor. Upon completion, one more token is added to P_{6-i}. The number of tokens is raised to 3, indicating the completion of the initialization of that processor. The processor itself enters P_{8-i}, where it executes the dispatched idle task.

t_7, P_4, t_8 and P_2 make the wait loop, where the processor 0 waits for the completion of all other processors' initializations. The waiting time is implemented by setting a counter to a big value, then decrementing it by one till 0. t_9 won't be enabled until t_6 does. However, t_6 is only enabled after all other processors have completed their initializations. Finally, t_9 fires, dispatches the first user task for processor 0. The system enters the normal processing state.

Note that if we drop the PN implementation for two wait loops, the correct execution sequence can be still assured. It is the matter of how accurate we want the model to be.

Chapter 5 Interrupt Handling

Seven levels of interrupt priorities are provided in MC68010 which is one of hardware realizations of Harmony. Below are their assignments in Harmony:

level 1 interrupt priority (Harmony priority 3)

level 2 interrupt priority (Harmony priority 2)

level 3 interrupt priority (Harmony priority 1)

level 4 interrupt priority (Harmony priority 0)

level 5 interrupt priority (interprocessor communication)

level 6 interrupt priority (used restrictively)

level 7 interrupt priority (not maskable)

Level 7 is the highest priority. Level 6 is used in primitives _Disable() and partly in _IP_int() only. Level 5 is assigned to primitive _IP_int(). Interrupts are inhibited for all priorities less than or equal to the current processor priority contained in the status register.

There are two interrupt modes used in Harmony: direct and transparent [23]. They both require that the processor enters the exception processing state, and the second level handlers are both written in assembler.

5.1 Direct Interrupt Mode

In this mode the interrupt comes directly from the hardware device physically connected to the processor. It is used when a task wants to do some I/O. That task sends a request to the server. The server/notifier task initiates an I/O command to the device, and blocks itself while waiting for an interrupt which signals the completion of the command from the device.

The calling graph is shown in Figure 5.1. The function descriptions for each element are as following:

_Await_interrupt(interrupt_id, rply_msg) : after issuing an I/O command, the controlling task calls this primitive, blocks there while waiting for an interrupt. Besides being an id of interrupt, the interrupt_id is also the byte offset into the vector _Int_table and _Dev_data_table. The rply_msg is a pointer to the volatile data which are from peripheral devices and must be captured when the interrupt is first detected.

_Disable() : written in assembler, changes the current processor priority to interrupt level 6, thus disables all interrupts.

_Enable() : in assembler, changes the current processor priority to the active task—caller's priority.

_Block() : in assembler, removes the caller from its ready queue and dispatches the next ready task. Returns when the blocked task becomes ready again.

The first level interrupt handler : hardware mechanism, receives hardware interrupt, saves the registers on the stack of the interrupted task, invokes a proper

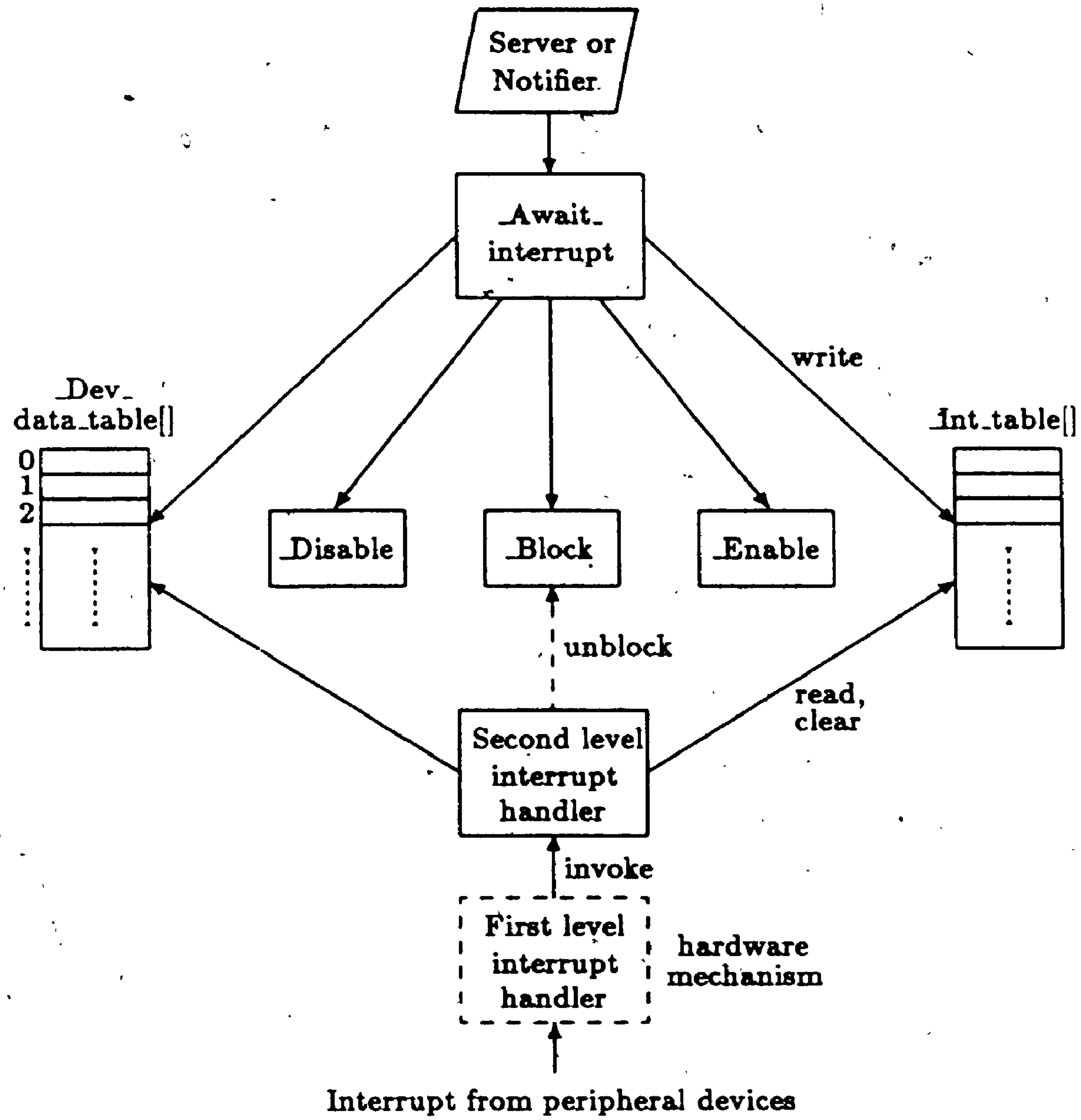


Figure 3.1 Calling Graph of Direct Interrupt Mode

second level interrupt handler.

The second level interrupt handler : written in assembler and server dependent. Its functions include saving stack pointer, etc., as will be detailed in Table 5.1 later. It must be initialized by the vector

struct INT_PAIR _Interrupt_list[] =

of all interrupt handlers required for a processor in user's Harmony program.

_Int_table[] : a data structure at which a pointer to the task descriptor of the waiting task is stored.

_Dev_data_table[] : a data structure, used to store a pointer to any data shared between the second level interrupt handler and the waiting task.

Now let's look at the algorithm of handling direct interrupt (also refer to the PN model in Figure 5.2 and Table 5.1). After issuing an I/O command, a task, usually a server/notifier, calls _Await_interrupt(). There, its td (task descriptor, a pointer to a data structure) is written into _Int_table, a pointer to volatile data is set, etc. Then it calls _Block() to block itself while waiting for an interrupt, and dispatches the next ready task.

On the other hand, upon finishing the I/O service, the peripheral device sends an interrupt to the processor it is connected to. If the interrupt priority level is higher than the current processor priority, the running task is interrupted. The processor enters the exception processing state. The invoked first level handler activates a proper second level handler. The processor starts normal processing.

The second level handler, such as _Ptm_int, _Serv_ptmint, usually saves the stack pointer of the interrupted task (does not remove it from the ready queue), identifies the interrupt source and finds a waiting task from _Int_table

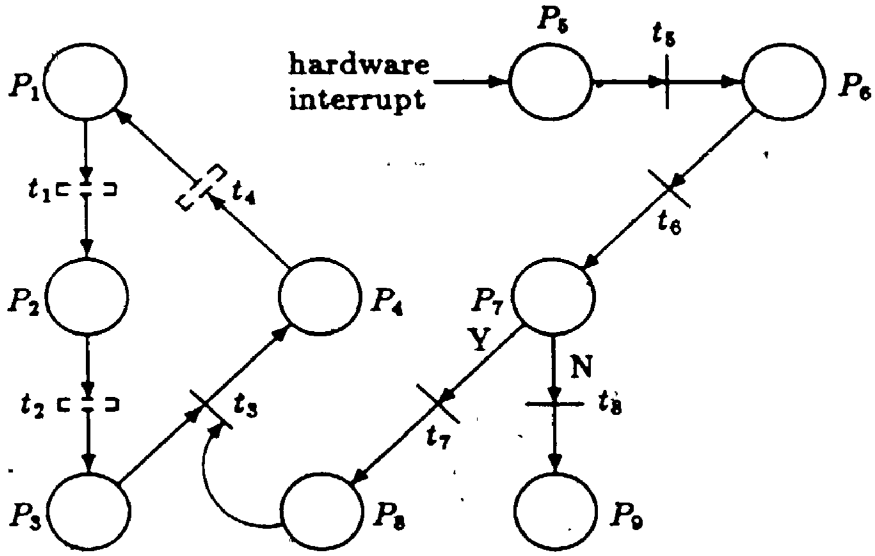


Figure 5.2 Direct Interrupt Mode

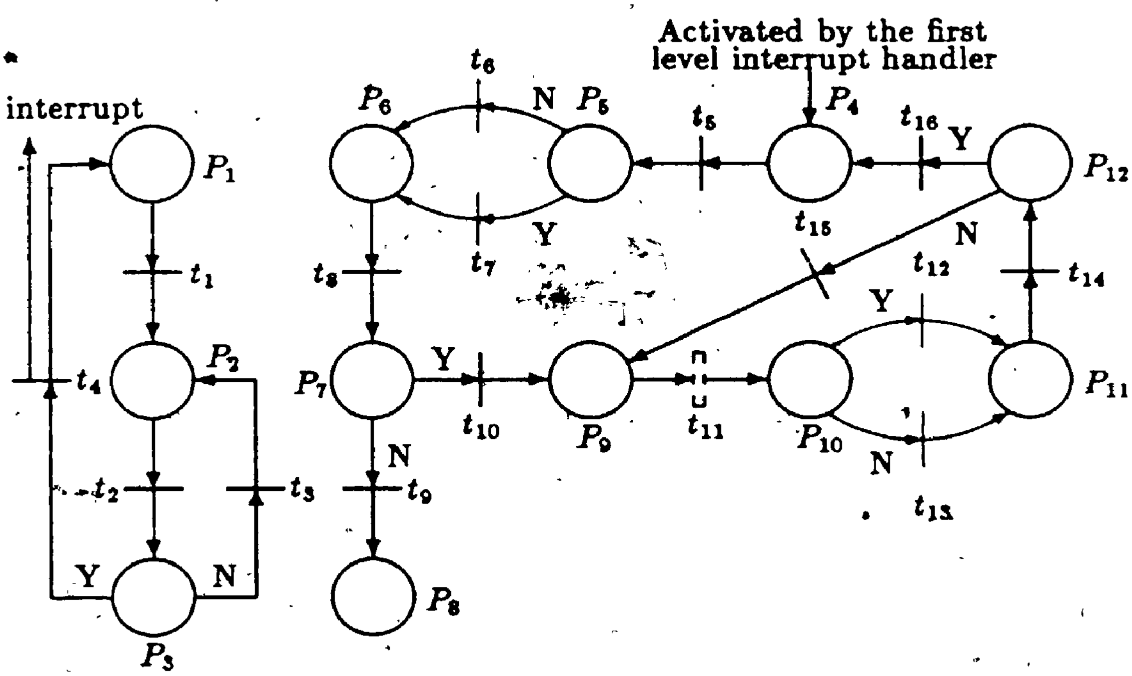


Figure 5.4 Transparent Interrupt Mode

Table 5.1 Direct Interrupt Mode

Trans.	Meaning
t_1	calls _Disable(), writes active task td to _Int_table, sets up some fields in id
t_2	calls _Block
t_3	restores stack and registers, activates waiting task
t_4	sets active task state to READY, calls _Enable()
t_5	saves registers of interrupted task, activates second level interrupt handler
t_6	saves stack ptr, identifies interrupt source, clears interrupt while records volatile data, gets the table entry and tests if there is one waiting task in it
t_7	clears table entry, passes volatile data, links the waiting task into the ready queue
t_8	increments _Spurcount, records cause, reactivates interrupted task
Place	**Meaning**
P_1	holds _Await_interrupt
P_2	the caller of _Await_interrupt is ready to block itself
P_3	the caller is blocked
P_4	the waiting task is unblocked
P_5	holds first level interrupt handler
P_6	holds second level interrupt handler
P_7	the second level handler is ready to test if there is a waiting task
P_8	the second level handler is ready to unblock the waiting task
P_9	holds interrupted task which is running

Table 5.2 Transparent Interrupt Mode

Transition	Meaning
t_1	saves status register, identifies destination processor
t_2	disables interrupt, executes test-and-set to find if mailbox slot is empty
t_3	provides interrupt window, reenters the busywait loop
t_4	writes id into mailbox, interrupts destination processor restores status register, returns
t_5	disables interrupt, saves registers and stack ptr, fetches id from my mailbox slot and puts it on a FIFO queue, turns off interrupt from interrupter, clears mailbox, advances queue index, needs to wrap queue index?
t_6	dummy transition
t_7	wraps producer index
t_8	is buffer full?
t_9	restores registers, resumes execution
t_{10}	replaces active task td by fake_td, gets new stack
t_{11}	enables writing _IP_int, calls _Td_service(), disables writing _IP_int, advances queue index, need wrap it?
t_{12}	wraps consumer index
t_{13}	dummy transition
t_{14}	is FIFO queue empty?
t_{15}	fetches id from queue
t_{16}	dispatches next ready task with the highest priority

Place	Meaning
P_1	holds _Signal_processor or _Block_signal_processor*
P_2	the busywait loop entrance
P_3	fork place for testing of the mailbox
P_4	holds _IP_int
P_5	fork place for testing of the buffer
P_6	producer index of the queue has been properly advanced
P_7	fork place for testing of the buffer
P_8	holds the interrupted task which is running
P_9	the loop entrance for calling _Td_service()
P_{10}	fork place for testing of the buffer
P_{11}	consumer index of the queue has been properly advanced
P_{12}	fork place for testing of the FIFO queue

* Minor changes are needed in this table for _Block_signal_processor

and activates it. If no waiting task has been found, it records such a spurious interrupt [3, p17] and reactivates the interrupted task.

5.2 Transparent Interrupt Mode

This interrupt mode is used for intertask communication. Tasks may run on different processors, or on a single processor. The interrupt handler (_IP_int) runs at interrupt priority level 5 which is above all other Harmony priorities. Therefore it is masked off from other interrupts and is transparent to the interrupt originator.

The calling graph is depicted in Figure 5.3, where the following elements are involved:

_Signal_processor(id) : called to notify processor of task requiring service.

_Block_signal_processor(id) : removes active task from its ready queue, sends an interrupt to destination processor, dispatches the next ready task.

_IP_int() : a second level interprocessor interrupt handler. Fetches id of a task requiring service and calls _Td_service() to serve the request.

_Td_service(id_candidate) : processes a td requiring service in message passing and task creation/destruction.

_Mailbox : a data structure with each slot assigned to one processor, where the id of the task requiring service is put. There is only one mailbox in the system.

_Comdev : similar to the above, it is used to store interrupt signal.

_In_id_q : is a FIFO queue (a circular buffer) for each processor. It holds id's

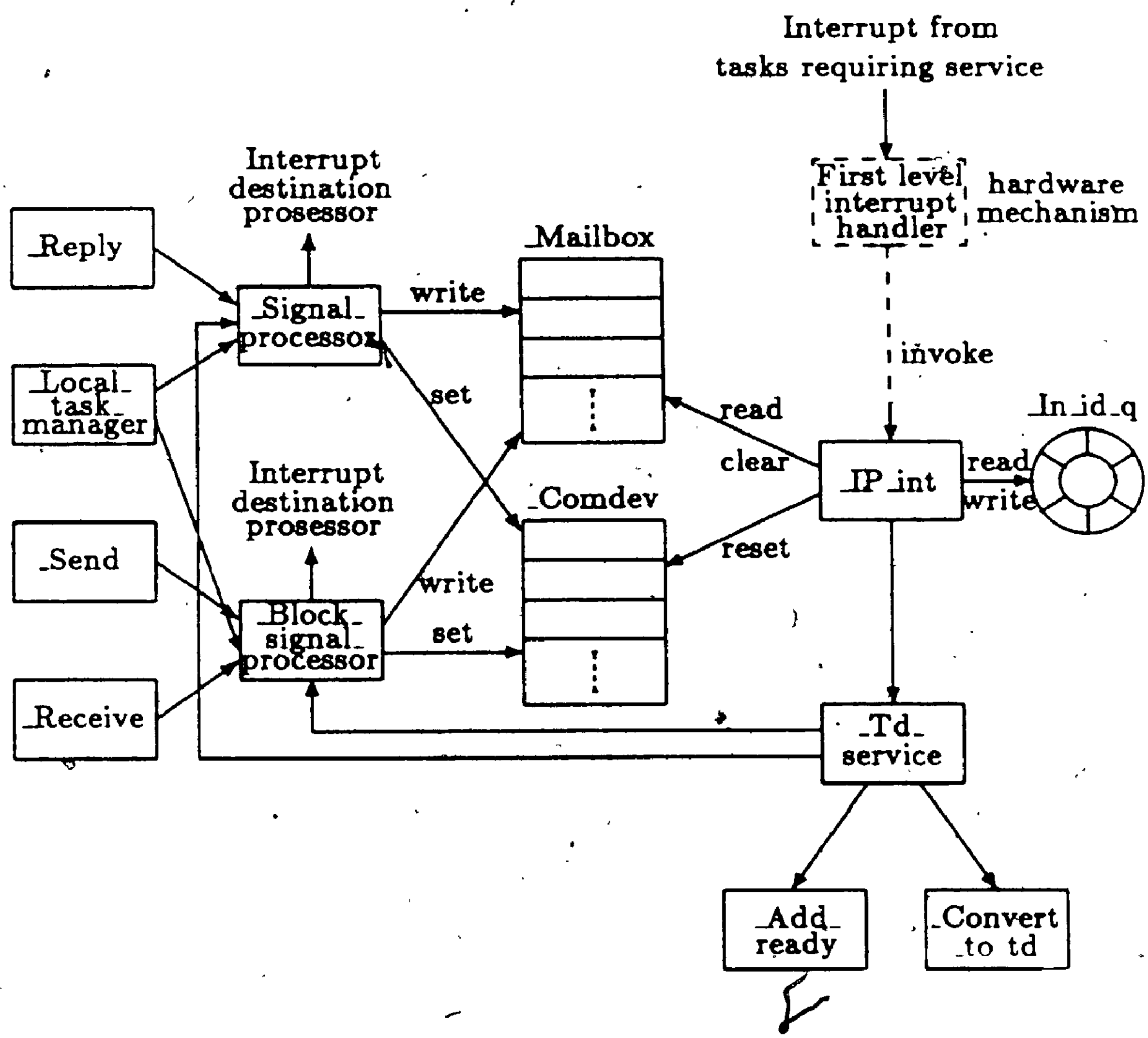


Figure 5.3 Calling Graph of Transparent Interrupt Mode

fetched from the mailbox.

Referring the PN model to Figure 5.4 and Table 5.2, now let's discuss how this interrupt mode works. When a task requires some services done on a task descriptor, such as changing task state, manipulating its queues, etc., it resorts to this interrupt mode. Actually such requests only arise from message passing, as well as task creation and destruction. So primitives _Send(), _Receive(), _Reply() and root function _Local_task_manager() make up the interface entity to the td service providing system. They call either _Signal_processor() or _Block_signal_processor() to inform the destination processor of their requests.

Because one mailbox slot can only hold one task id at a time and more than one processor may compete for accessing the slot at the same time, some kind of arbitration is necessary in gaining access to a slot of the mailbox. The pre-busywait (check before put) is employed in Harmony.

Tasks poll the mailbox by executing the test-and-set instruction at t_2 in Figure 5.4. This guarantees that only one task can access a single mailbox slot. Before testing, all interrupts are disabled. If the mailbox is not empty, that is, the previous task has not acquired td service yet, the polling task will change the current processor priority to level 4, thus provides an interrupt window for the interrupt handler _IP_int to fetch and clear the mailbox. Without this, the polling task will hog the mailbox, the mail will never be taken by _IP_int. It is a deadlock.

After writing the id into the mailbox, _Signal_processor sends an interrupt to the destination processor through using _Comdev table. The properly invoked handler _IP_int runs at processor priority level 6. It fetches the id from the mailbox then clears it at t_5. The fetched id is then put on a FIFO queue (a circular buffer). At t_8 the queue is examined to see if it is full. If the buffer is full, the handler fetches one item, and calls _Td_service() by specifying this id as passed in argument. Then the handler checks the FIFO queue again until has all tasks served.

Note that before calling _Td_service() at t_{11}, the handler changes the processor priority to level 4, that enables putting the new id to the FIFO queue in the following way. Suppose at this moment the processor detects an interrupt from _Signal_processor, it will preempt the current executing _IP_int at level 4 by activating a new instantiation of _IP_int to fetch id from the mailbox and put it on the FIFO queue at level 6.

After served all requests, _IP_int dispatches the next ready task at t_{16}.

Chapter 6 Message Passing

Harmony is a multitasking operating system. Sometimes, the tasks need to communicate and synchronize with each other. This is referred to as the message passing.

In this chapter, we first examine the algorithms used. Then to get a quick impression, we present two simple models at the high level. Afterwards we get down to the low level models with emphasis on a pair of communicating tasks. To model the communications between multiple tasks, the double colored token is introduced to our final model. The case of multiple communicating tasks and deadlock are finally examined. A useful reference is [8].

6.1 Descriptions of the Algorithms

Message passing in Harmony is implemented by four primitives:

```
id = _Send( rqst, rply, id );

id = _Receive( rqst, id );

id = _Try_receive( rqst, id );

id = _Reply( rply, id );
```

The semantics of these functions are as following [3, p9] : to send a message to another task, a task first sets up that message in space pointed to by the rqst argument, then sets up space pointed to by the rply argument into which the replied message will be saved, and finally calls the _Send primitive with the id of the desired correspondent task specified in the id argument. To receive a message, a task first sets up space pointed to by the rqst argument into which the contents of the sender's rqst message can be copied, and then calls one of receiving primitives with the id argument specified either as the id of a particular correspondent task or 0, the latter representing receiving a message from any task. After finishing processing the sender's request, the receiving task sets up the message to be replied in space pointed to by the rply argument, and then calls the _Reply primitive with the id argument specified as the sender.

The calling graphs for each primitive are depicted in Figure 6.1 through 6.4. In Figure 6.1 and 6.2, the function _Signal_processor activates _IP_int more than once. The functions involved but having not been introduced so far are listed as follows :

_Add_ready(td) : puts the calling task on its ready queue.

_Convert_to_td(id) : converts a given id into a task descriptor (td).

_Copy_msg(from, to) : copies "from" string to "to" string.

_Receive(rqst_msg, id) : called when a task is ready to receive messages. The task may specify whether it wants to receive messages from a specified task or any tasks.

_Reply(rply_msg, id) : replies a message to the task with the id.

_Send(rqst_msg, rply_msg, id) : sends a message to the task with the id specified in the call.

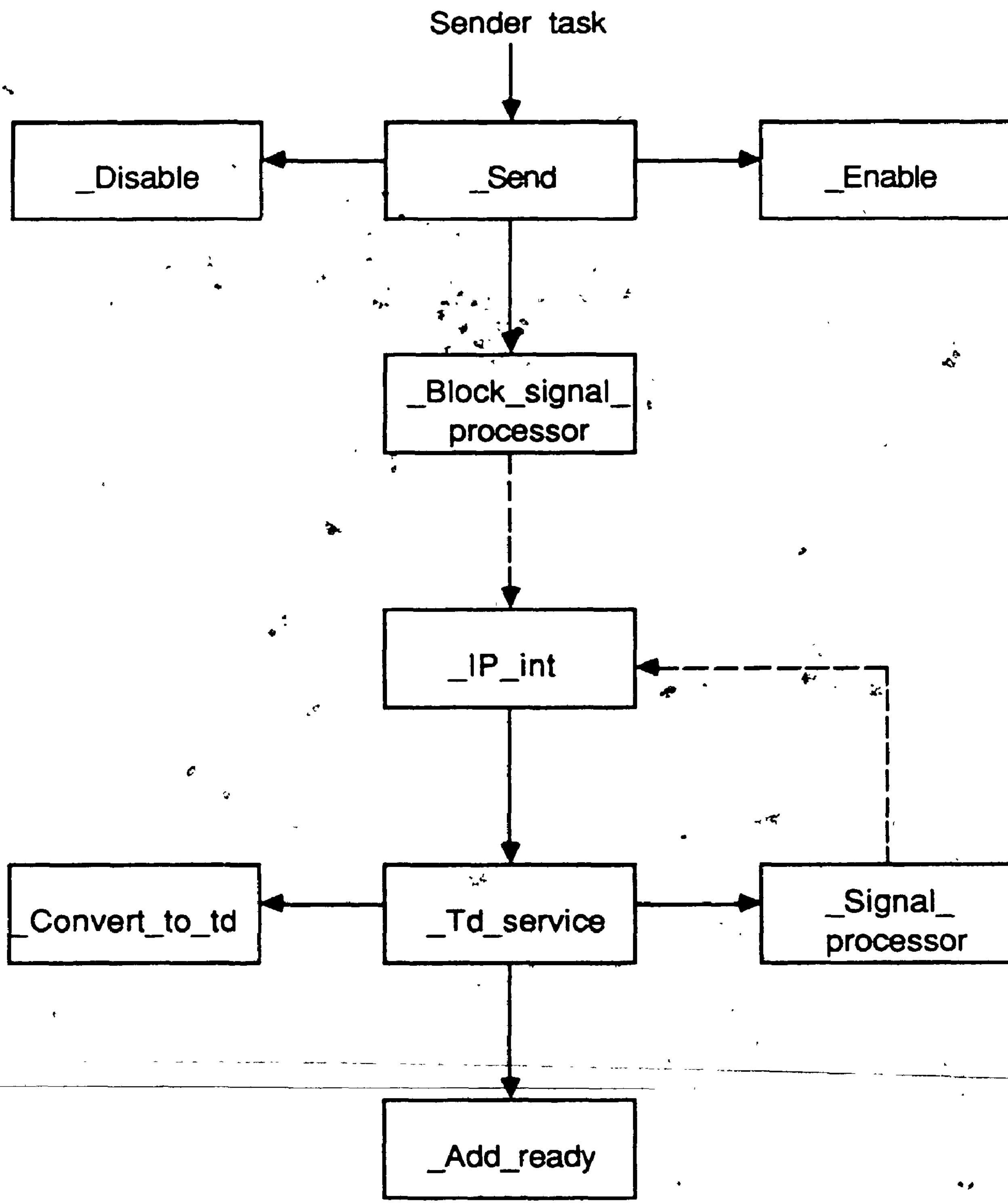


Figure 6.1 _Send( )

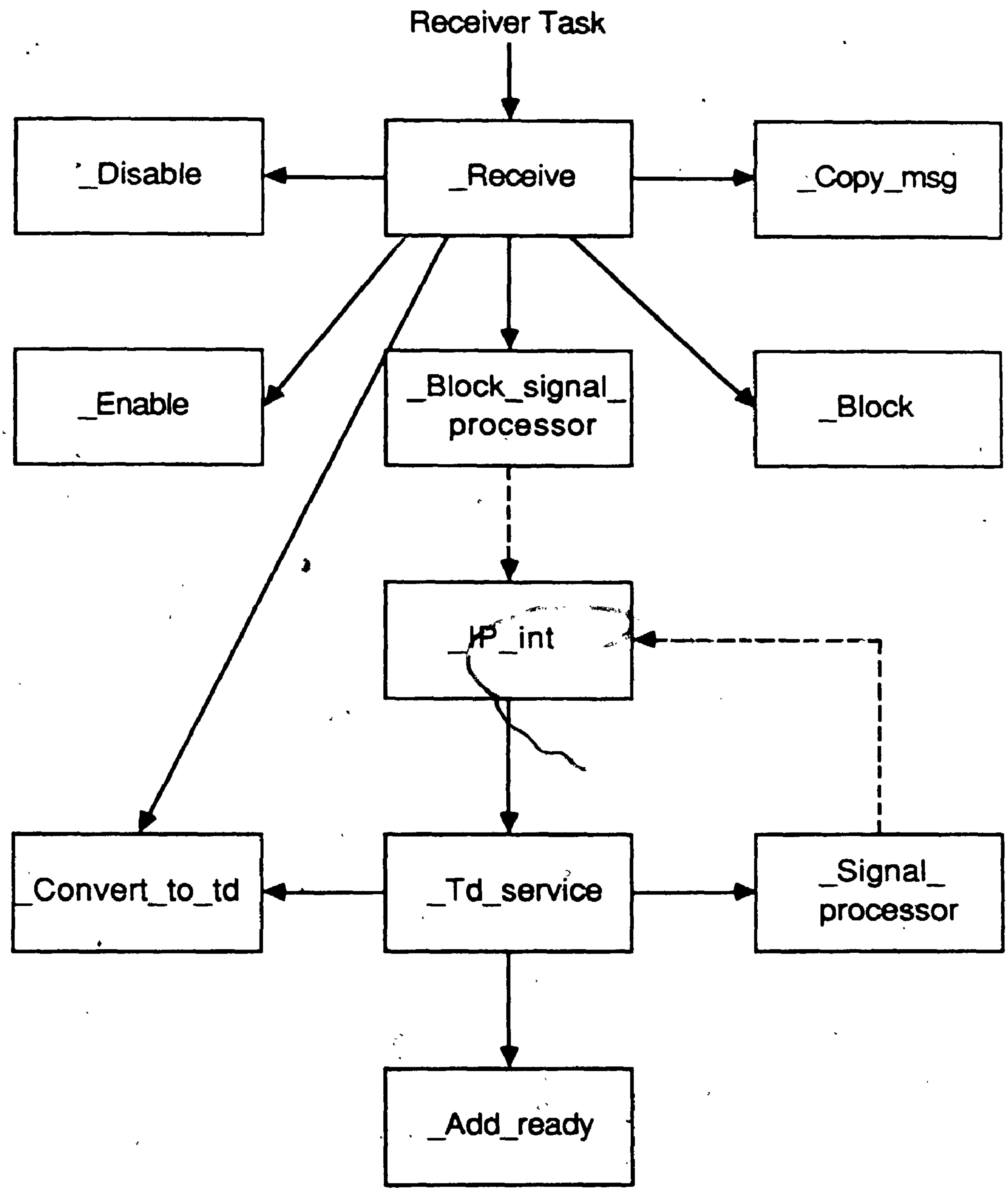


Figure 6.2 _Receive( )

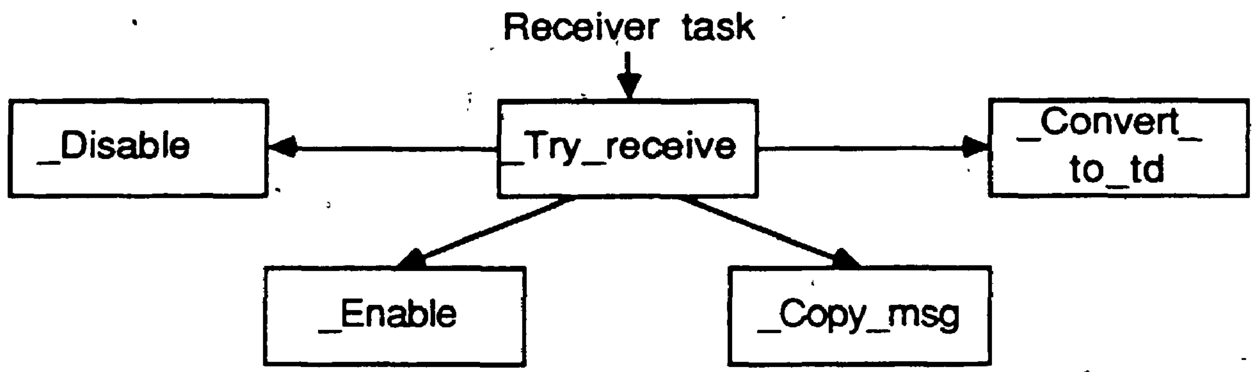


Figure 6.3 _Try_receive( )

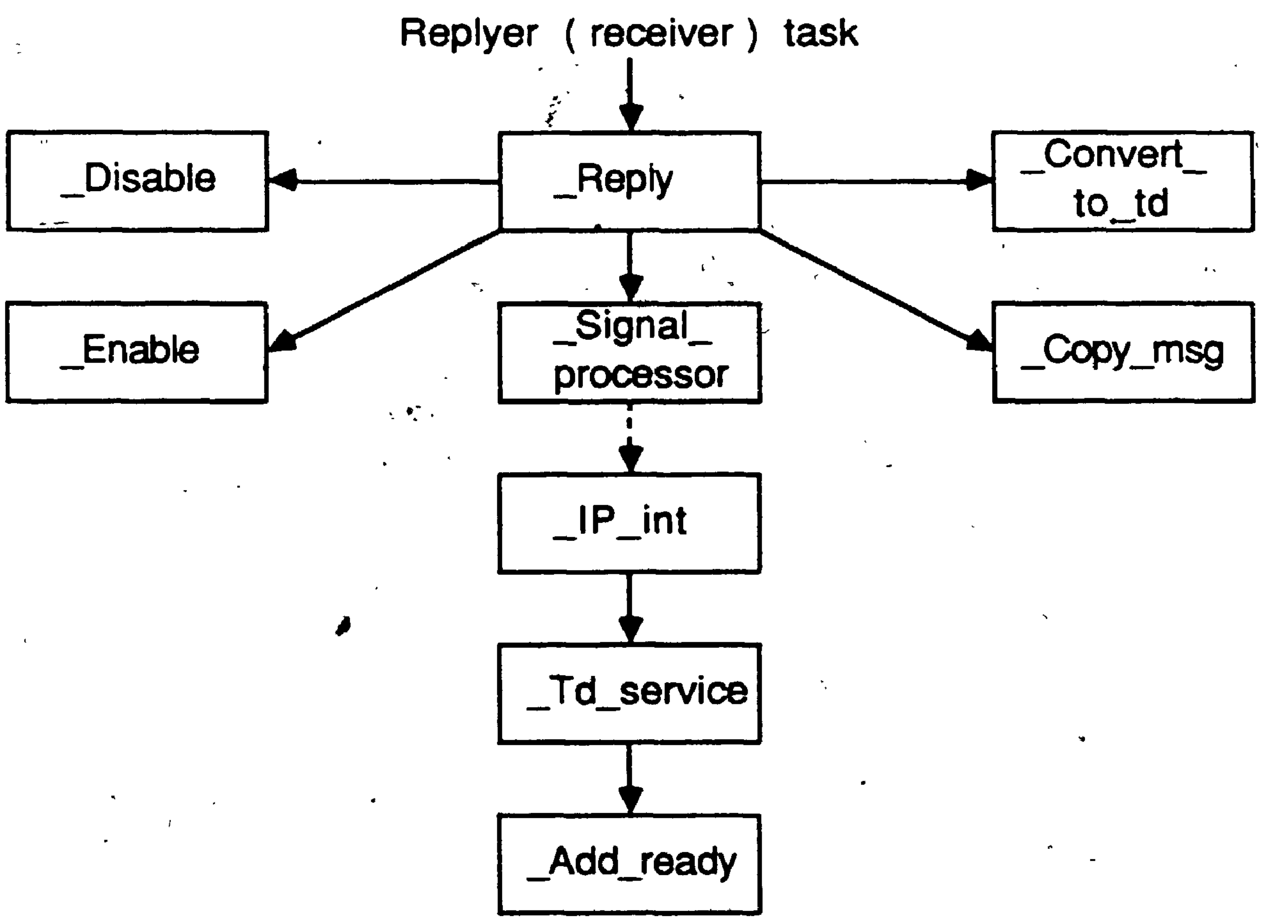


Figure 6.4 _Reply( )

_Td_service(id_candidate) : processes a td requiring service.

_Try_receive(rqst_msg, id) : receives a message from the sender task specified by the id without blocking itself.

Each task is associated with a data structure called task descriptor (td). Among the fields, some are frequently used in message passing. The field STATE is for keeping record of the state of a task. The field CORRESPONDENT is for saving the id of a correspondent task. The REQUEST_MSG is a pointer to a character type array for a sender into which the message to be sent is saved. The REPLY_MSG is a pointer to character type array for a sender into which the replied message from the receiver is saved.

Among queues maintained by a task through its td, the send_q, receive_q (recv_q) and reply_q are used for message passing, as depicted in Figure 6.5 where the recv_q is empty. The send_q belongs to a receiver task and is used for senders. So does the reply_q. The td's of those sender tasks who are blocked in sending a message to this receiver task are FIFOed in this queue. The recv_q belongs to a sender task and is used for receivers. In the primitive _Receive() from the specific, the td of the receiver is put on the recv_q at the sender when a message from a specific sender is expected to be received and upon the receiver's state advances to the ACK_Q_RECEIVER. In _Receive() from the any, the td of the receiver is never placed on the recv_q. The td of the sender is put on the reply_q at the receiver before message copying starts.

There is a rule for manipulating these queues. That is, a queue can only be manipulated on its owner task's processor, because tasks are tied to their specific processors. In other words, to put the td on one of the above three

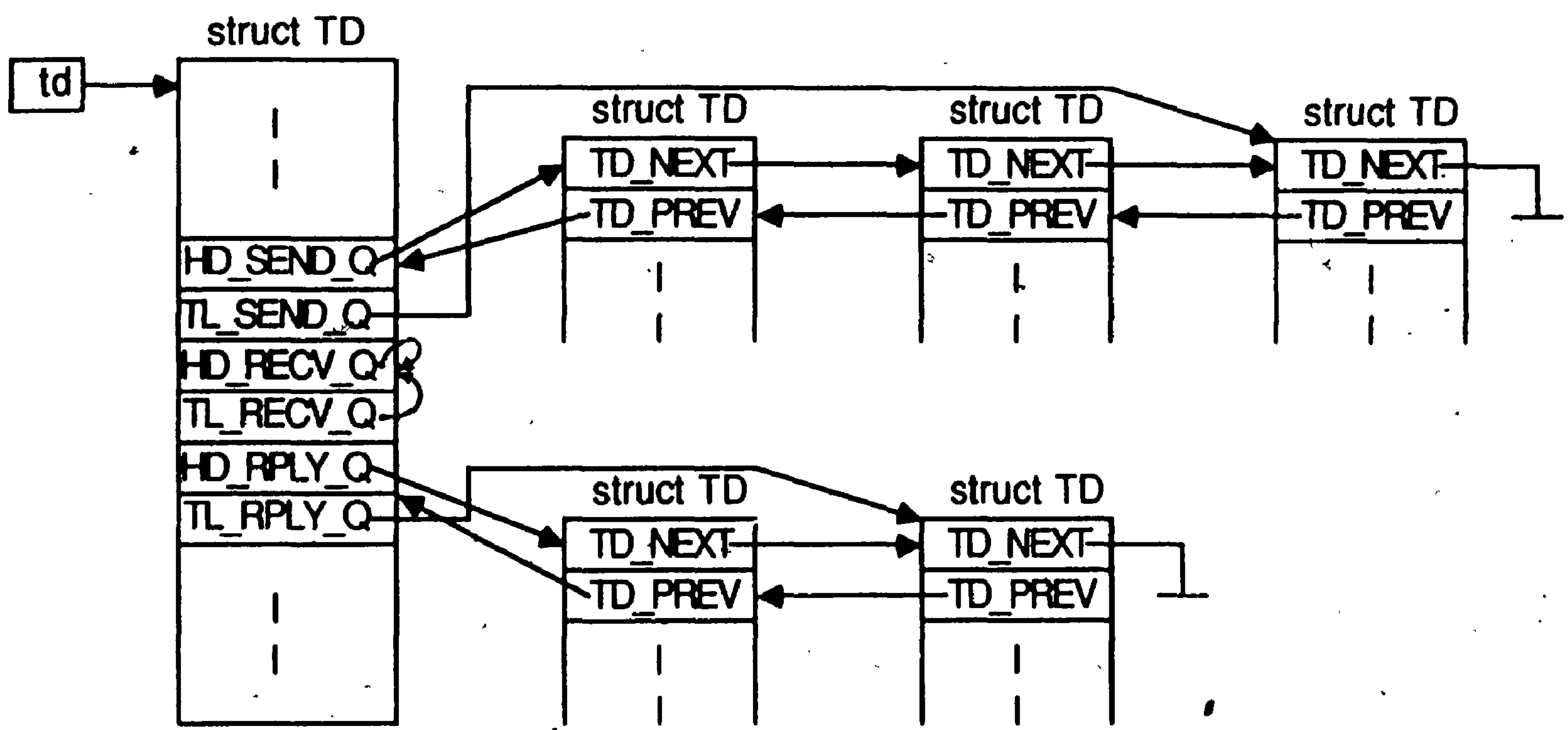


Figure 6.5 Three Communication Queues Maintained through a Task Descriptor (td)

queues, the queuing operation has to be done on the processor where the owner of that queue resides. To ensure this rule, the transparent interrupt mode is extensively used to pass such td service requests around. The function _Td_service is a place where the actual queuing operations take place.

As to the blocking behavior of these primitives, the _Send() and _Receive() are blocking primitives, the other two are nonblocking ones. This aspect of the semantics can be efficiently expressed by the high level Petri nets models in the next section.

6.2 High Level Petri Nets Models

A Petri Net model for message passing in the high level is depicted in Figure 6.6 and Table 6.1. This model gives us a flavor of how task communication algorithm works.

Case 1 (sender activated first):

a) Receiver not activated:

The sender is available by a token in place P_1. Firing t_1 represents that the sender calls primitive _Send(). Then one token is put into P_2, i.e., the sender blocks itself until unblocked by returning of _Reply(). Meantime, a control token moves into P_{10}.

b) Receiver activated later:

The receiver has the choice of calling either _Receive() or _Try_receive().

If _Receive() is called, only t_9 is enabled and fires. Then it might call _Reply(), ..., t_4 fires, _Reply() returns. The receiver returns to the original place P_{13}, and t_4 sends a token to P_3 so that

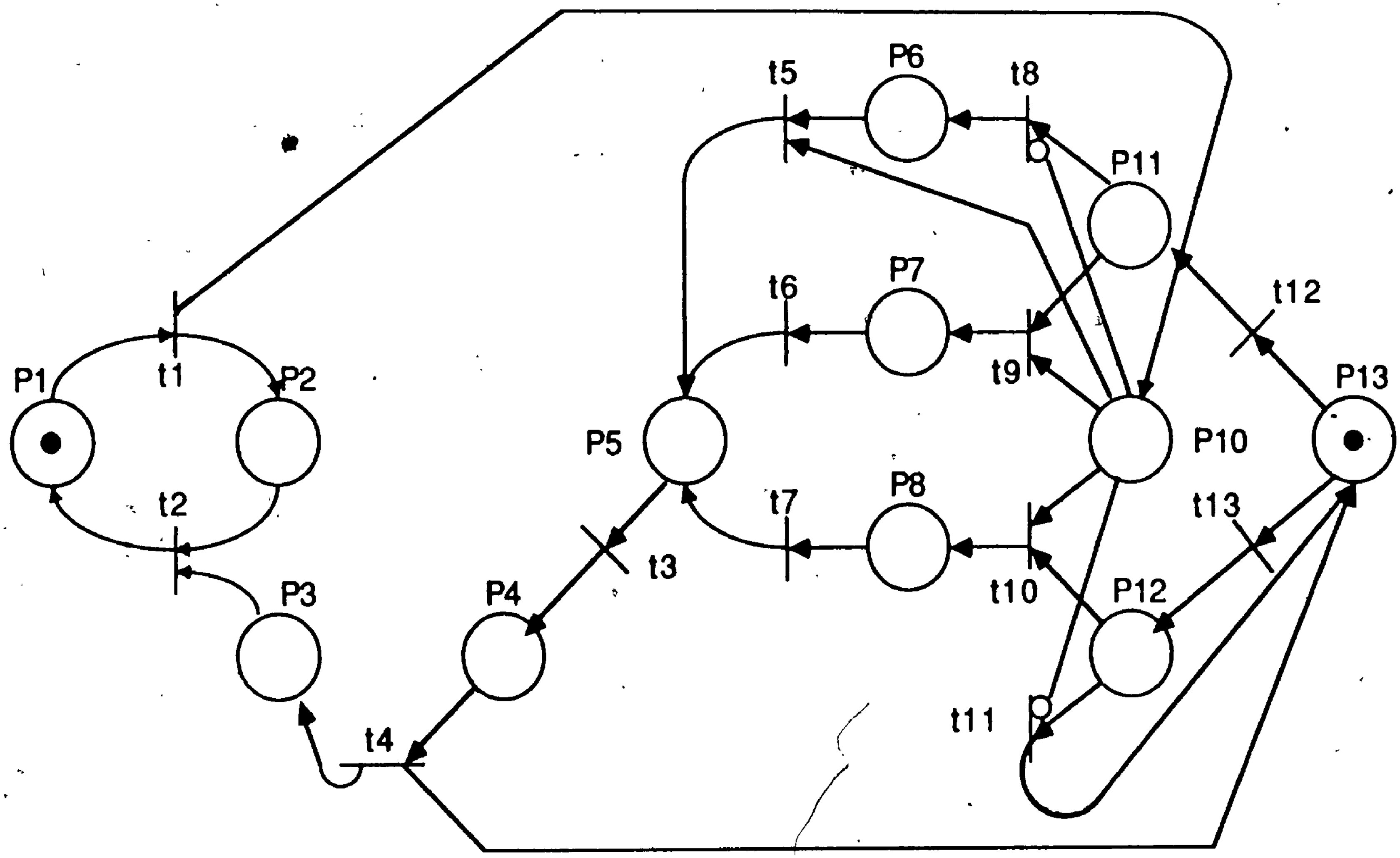


Figure 6.6 A Simple Overall Model

Table 6.1 A Simple Overall Model

Transition	Meaning
t_1	calls _Send()
t_2	sender is unblocked, _Send returns
t_3	calls _Reply()
t_4	_Reply sends a token to unblock the sender, meantime it returns
t_5	blocked receiver is released by activation of _Send(), _Receive returns
t_6	_Receive() returns
t_7	_Try_receive returns
t_8	receiver blocks itself due to sender not available
t_9	_Receive proceeds due to sender available
t_{10}	_Try_receive succeeds due to sender available
t_{11}	_Try_receive fails due to sender not available
t_{12}	calls _Receive()
t_{13}	calls _Try_receive
Place	**Meaning**
P_1	initial place for a sender
P_2	sender is blocked
P_3	holds a control token which is going to unblock sender
P_4	_Reply in activation
P_5	temporary place for receiver after it has received a message and before it calls _Reply
P_6	receiver is blocked
P_7	receiver is in progress at the non-blocking branch
P_8	_Try_receive is in progress
P_9	not used
P_{10}	holds a control token which indicates activation of sender
P_{11}	_Receive was called
P_{12}	_Try_receive was called
P_{13}	initial place for a receiver

the sender is released.

If _Try_receive() is called, the same story will happen.

Case 2 (receiver activated first):

a) Sender not activated:

If the receiver calls _Receive(), t_8 is enabled, t_9 not. The receiver blocks itself at place P_6 until _Send() is called and t_5 is enabled. t_8, P_6 and t_5 may be considered redundant. They are there because first we want to make P_6 as a blocking place explicitly. Secondly, this branch is a symmetry of the failure branch in _Try_receive(). When doing performance analysis, this branch can be removed.

If the receiver chooses _Try_receive(), since t_{11} is enabled among the pairs: t_{10}, t_{11}, _Try_receive() fails immediately.

b) Sender activated later:

For upper route _Receive(), the receiver blocked at P_6 will be released when _Send() is called. Then the receiver might call _Reply() to unblock the sender.

For the next route _Try_receive(), the receiver fails before _Send() is called.

To simulate the situation an active task faces more closely, a revised natural model can be obtained as depicted in Figure 6.7 and Table 6.2. A task at P_1 has four choices. The comparison is tabulated below.

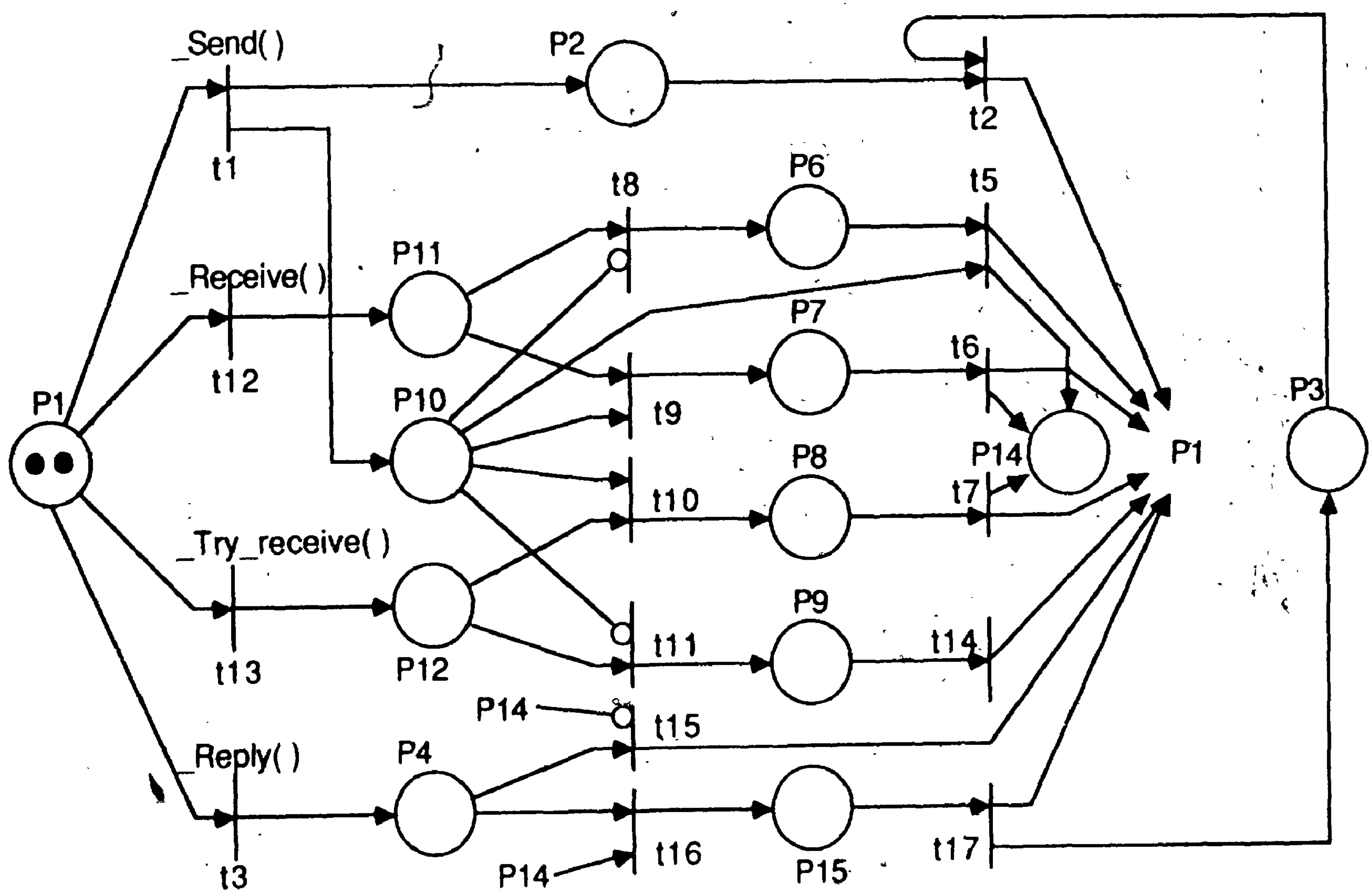


Figure 6.7 A Natural Overall Model

Table 6.2 A Natural Overall Model

Transition	Meaning
t_1 — t_{13}	have the same meaning as in Table 6.2 if not default
t_{14}	_Try_receive returns a 0 due to failing
t_{15}	_Reply fails as the caller had not ever received a message before calling _Reply()
t_{16}	_Reply succeeds because a message had been received before
t_{17}	_Reply returns
Place	**Meaning**
P_1	initial place for communicating tasks
P_2 — P_8	have the same meaning as in Table 6.2 if not default
P_9	_Try_receive is in progress at the failing branch
P_{10} — P_{13}	have the same meaning as in Table 6.2 if not default
P_{14}	holds a control token which indicates a message has been successfully received by the receiver
P_{15}	_Reply is in progress at normal branch

Table 6.4 The First Low Level PN Model

(To be continued on page 53)

Place	Meaning
P_1 — P_9	places for receiver's states
P_{10} — P_{16}	places for sender's states
P_{17}, P_{18}, P_{24}	are irregularly defined as the caller's correspondent's correspondent is the caller by a token in it without concerning how the token moving in
P_{19} — P_{23} and P_{25}	drawn in small circle are places for control tokens, tasks do not enter them

Table 6.3 The Comparison Between Two Models

Calling Sequence	Models	
	Simple	Natural
S, Rc,Rp	Y	Y
S, Rp,Rc	N	Y
Rc,S, Rp	Y	Y
Rc,Rp,S	N	Y
Rp,S, Rc	N	Y
Rp,Rc,S	N	Y

Where, for instance, the row "Rc, Rp, S, N, Y" means that the calling sequence is _Receive() or _Try_receive(), _Reply() and _Send(); the simple model fails to represent this sequence, however the natural model can.

In next section, we introduce four detailed models in the order from simple to complex. The natural firing sequence identical to the flow of control in primitives is tracked to explain models.

6.3 Low Level Petri Nets Models

In following figures, the states of tasks are borrowed from their definitions in Harmony source code except ACTIVE. No more task states are defined. Small circles are used for control tokens only. The colored net is used as well. Some places (states) belong to receivers, others to senders. Receivers or senders flow along their own arcs. Control tokens behave in the same way.

The dashed place, like place P_{18} in Figure 6.8, is defined irregularly as "my correspondent is trying to communicate with me" by a control token in

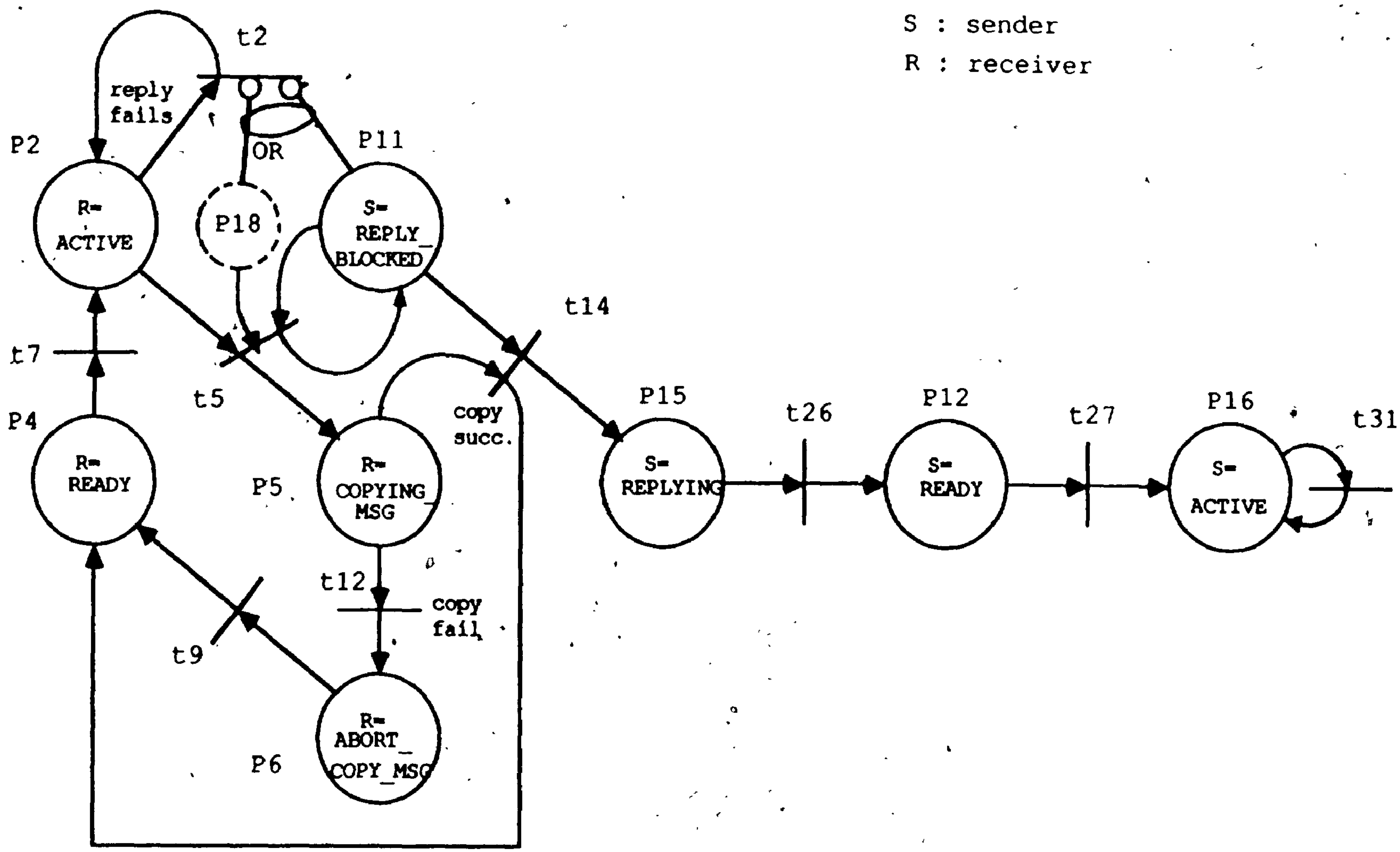


Figure 6.8 Reply()

it without concerning how the token moves in. Transition t_2 is defined as "OR-AND" transition which is actually an abstraction of a functional subnet.

6.3.1 _Reply(rply_msg, id)

Referring the Petri net model to Figure 6.8 and Table 6.4, upon a call to this function, the sender is supposed to have sent a message to the receiver and stays at the state REPLY_BLOCKED.

Case 1 (normal case):

There is one token in P_2, P_{11} and P_{18} each for initial states. When the receiver calls _Reply(), the sender's (replyee's) state equals "REPLY_BLOCKED", and it is blocked for me due to a call to _Block_signal_processor() earlier. t_5 is enabled, and fires, removing the sender from the reply_q at the receiver. The receiver now is copying message. Both t_{12} and t_{14} are enabled. If the sender is still alive, t_{14} fires, t_{27} follows, adds the sender to the ready_q, unblocks the sender. If copying fails, t_{12} fires, the receiver backs to the READY state, then to the ACTIVE and returns 0 to indicate that _Reply() has failed.

Case 2 (abnormal):

Initially, t_2 is enabled instead of t_5. It fires, returns 0 and backs to the ACTIVE. _Reply() fails immediately.

6.3.2 _Try_receive(rqst_msg, id)

The model is given in Figure 6.9 and Table 6.4. This is an unblocking primitive.

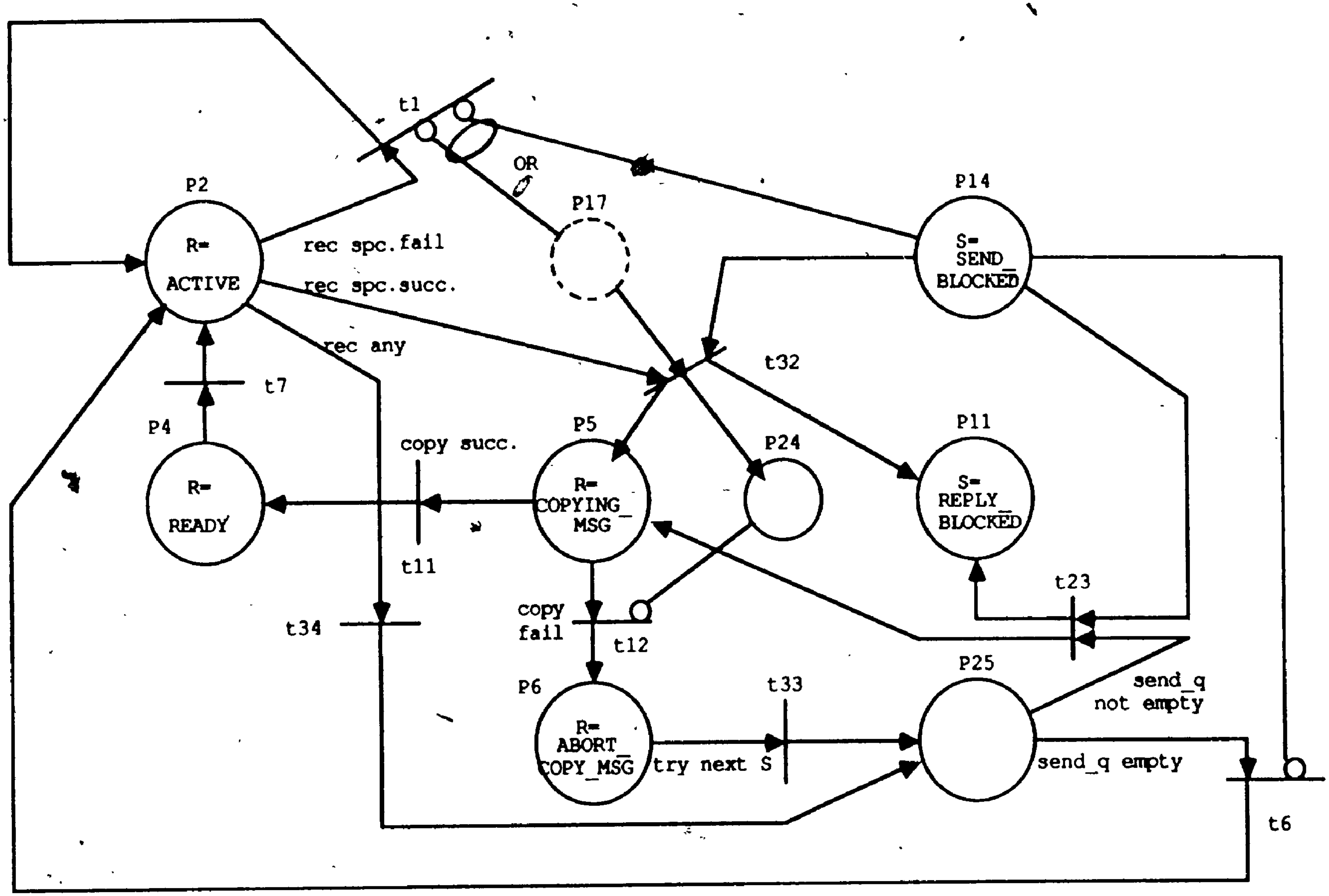


Figure 6.9 _Try_receive( )

Table 6.4 The First Low Level PN Model
(Continued from page 48)

Trans.	Meaning
t_1	_Try_receive fails because the sender is dead or the caller's send_q is empty or other reason as illustrated
t_2	_Reply fails because the replyee is dead or other reasons as illustrated
t_3	calls _Receive specific, blocks the caller, signals correspondent's processor, dispatches the next ready task
t_4	unblocked caller of _Receive specific fails because its correspondent is dead
t_5	calls _Reply, removes the sender from reply_q at the receive
t_6	_Try_receive any fails due to empty send_q at the receive
t_7	dispatches a ready task
t_8	adds the receiver onto recv_q at sender as the sender alive
t_9	_Reply fails because copying fails
t_{10}	dvances receiver's state as the sender is dead
t_{11}	copies sender's message to receiver successfully
t_{12}	copying message fails
t_{13}	calls _Receive any, blocks itself as its send_q empty, dispatches a ready task
t_{14}	copies receiver's replying message to the sender successfully
t_{15}	advances receiver's state
t_{16}	unblocks receiver by adding it to the ready_q as sender is dead
t_{17}	caller of _Receive specific is unblocked and enters COPY_MSG state as the sender is alive
t_{18}	caller of _Receive any blocks itself dispatches another ready task as its send_q is empty
t_{19}	removes receiver from recv_q at sender when receiver is alive
t_{20}	the sender dies
t_{21}	adds sender to reply_q at receiver, unblocks receiver by adding it to the ready_q when sender is alive
t_{22}	removes sender from send_q at receiver when sender is alive and on send_q
t_{23}	OR-AND transition, when _Receive any or _Try_receive any called, removes sender from send_q at receiver, adds it to reply_q at receiver when _Try_receive specific called and the caller is sender's correspondent, advances sender's state only
t_{24}	blocked receiver is unblocked by a sender becoming available
t_{25}	OR-AND transition, adds sender to send_q at receiver as receiver is alive
t_{26}	unblocks sender by adding it to its ready queue
t_{27}	dispatched
t_{28}	sets up message pointers, blocks sender itself, signals correspondent's processor, dispatches the next ready task
t_{29}	advances sender's state due to dead receiver
t_{30}	calls _Send
t_{31}	unblocked sender gets redispatched to return

Case 1 (try receiving specific):

Initially, there is one token in P_2. If the sender is not available, the receiver will fail immediately, return 0. If either P_{14} or P_{17} is empty or both empty, t_1 fires. _Try_receive() fails. If t_{32} is enabled, the receiver enters COPYING_MSG state, while the sender to P_{11} REPLY_BLOCKED state. In this primitive, P_{11} is the destination place for a sender. The sender will proceed further in _Reply() primitive. Notice that no queuing operations were taken during t_{32} firing. It may be assumed that the receiver will call _Reply() after _Try_receive() returns, because a task which calls _Try_receive() is very likely a hasty user. But it seems to be safer if the queuing operations "remove the sender from the send_q at the receiver" and "put the sender on the reply_q at the receiver" are inserted during the state transition. This is because the receiver may call some other functions before calling _Reply(). Moreover, in _Reply() primitive, "remove the sender from the reply_q at the receiver" is always present when the receiver moves into COPYING_MSG state. The receiver expects the sender in its reply_q when it calls _Reply().

Now t_{12} is disabled, t_{11} fires. Copying message succeeds. The receiver returns to state READY, then ACTIVE.

Case 2 (try receiving any):

t_{34} fires, the receiver enters P_{25}. If the send_q empty, i.e., P_{14} empty, the receiver backs to P_2, _Try_receive() fails. If the send_q not empty, t_{23} fires, one sender is removed from the send_q, then added to the reply_q. The receiver enters state COPYING_MSG. Now both t_{11} and t_{12} are enabled. If the sender is killed, copying fails, the receiver's state becomes

ABORT_COPY_MSG. Then it checks the send_q to see if any more senders there, until gets one.

6.3.3 _Receive(rqst_msg, id)

This is a blocking primitive. Refer the model to Figure 6.10 and Table 6.4.

Case 1 (receive any):

If the caller's send_q not empty, t_{23} fires, removes the sender from the send_q to the reply_q. The sender changes to state REPLY_BLOCKED, while the caller enters P_5 COPYING_MSG. Now both t_{11} and t_{12} are enabled. If the sender is killed, t_{12} fires, the token moves into P_6 ABORT_COPY_MSG. If the send_q not empty, the caller will try the next sender by firing t_{23}, repeat the cycle. If the send_q empty, t_{18} fires, the caller enters state RCV_BLOCKED, blocks itself, dispatches another task at the ready_q.

Later on, the blocked receiver at P_9 may be unblocked by the coming of a sender. t_{24} fires, the receiver back to state ACTIVE eventually. It checks the send_q to decide how to continue execution.

Case 2 (receive specific):

t_3 fires, blocks the caller, interrupts correspondent's processor. The caller enters state Q_RECEIVER.

a) Fast route, the sender was killed:

t_{10} fires, a token moves into P_8. Since the sender is dead, only t_{16} fires. It unblocks the receiver, adds it to the ready_q. After back to P_2 ACTIVE, t_4 is enabled. It fires, returns 0. _Receive( ) fails.

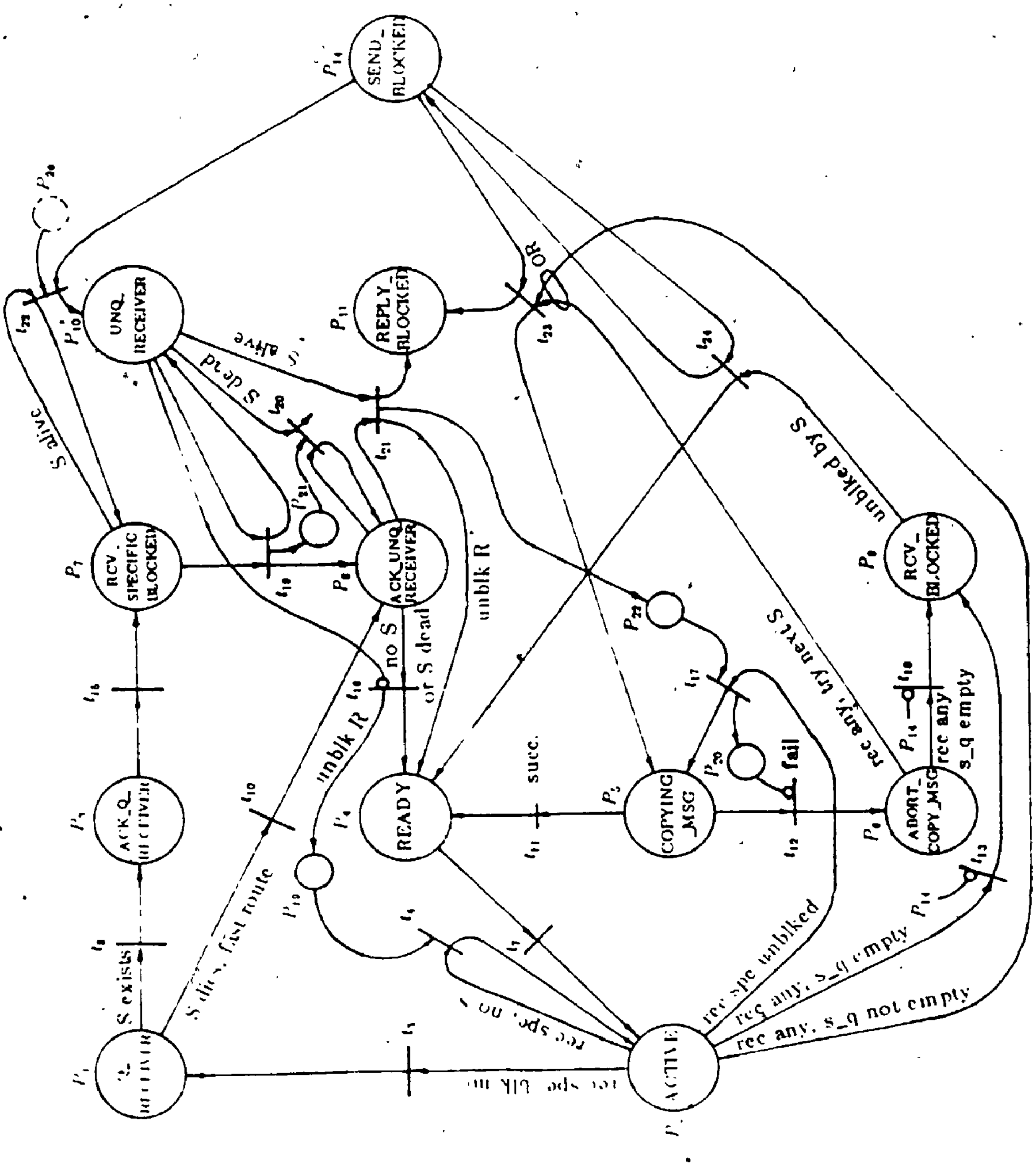


Figure 0.10 Receive()

b) Sender exists:

t_8 fires. The receiver is put on the reply_q at the sender. Later the token flows into P_7 RCV_SPECIFIC_BLOCKED. Now if t_{22} is disabled, so is t_{19}, the receiver will stay in P_7 for arbitrarily long. Otherwise t_{22} fires, the sender enters P_{10} UNQ_RECEIVER. Then if the receiver alive, t_{19} fires. The receiver comes to state ACK_UNQ_RECEIVER. At this time, if the sender dies, t_{20} fires, and t_{16} fires. It unblocks the receiver. Finally t_4 fires, the receiver returns. If the sender alive, t_{21} fires. The unblocked receiver will continue execution by firing t_{17}. Its state changes to COPYING_MSG. Since t_{12} is disabled now, only t_{11} fires. _Receive() succeeds.

Notice that if the receiver in P_7 RCV_SPECIFIC_BLOCKED dies, in the source code the receiver's state is still set to ACK_UNQ_RECEIVER and proceeds. Therefore, the fishy things arise. It's felt that, at this moment, something should be done on the sender, because it's already unqueued from the send_q, and at an unfavorable state UNQ_RECEIVER instead of SEND_BLOCKED. My idea for correction is that the sender unblocks itself, and returns 0 to indicate the failure of _Send(). Reason one, the particular correspondent of the blocked sender was dead and there makes no sense to try to send a message to nobody. Reason two, when we look at the _Send() primitive in the coming Section 6.3.4, it's learned that when the sender finds its correspondent—receiver not existing in the system, it will unblock itself immediately and return 0. So the newly designed code is given in Appendix B, which works for the _Send() primitive, as well.

6.3.4 _Send(rqst_msg, rply_msg, id)

This is a blocking primitive. The model is depicted in Figure 6.11 and Table 6.4.

A sender activates itself by firing t_{30}, then sets up pointers, blocks itself, interrupts the receiver's processor, dispatches another task. Receiving interrupt, the receiver's processor calls _Td_service() from _IP_int(). The sender enters P_{14} SEND_BLOCKED.

Case 1 (receiver dead):

t_{29} fires. The sender enters state REPLYING, then unblocks itself.

Case 2 (receiver exists but may neither try to get a message from this sender, nor at RCV_SPECIFIC_BLOCKED):

t_{25} fires. The sender is queued on the send_q at the receiver. If the receiver is not at state RCV_BLOCKED, t_{24} can not fire. The sender stays in P_{14} for arbitrarily long. If the receiver is at RCV_BLOCKED, t_{24} fires. It unblocks the receiver, which will in turn check whether the sender is dead later. If the sender dead, t_{13} fires. The receiver enters state RCV_BLOCKED again. The rest would be the same as the described in above subsection.

Case 3 (receiver exists at state RCV_SPECIFIC_BLOCKED and is trying to communicate with me):

t_{22} fires, the sender enters P_{10}. Then only t_{19} fires. The receiver is removed from the recv_q if it's alive and moves to state ACK_UNQ_RECEIVER. Now if the sender dies, t_{20} then t_{16} fires. The receiver is unblocked. Later t_4 fires, the receiver returns 0. If the sender alive, t_{21} fires. The sender goes to the REPLY_BLOCKED state and unblocks

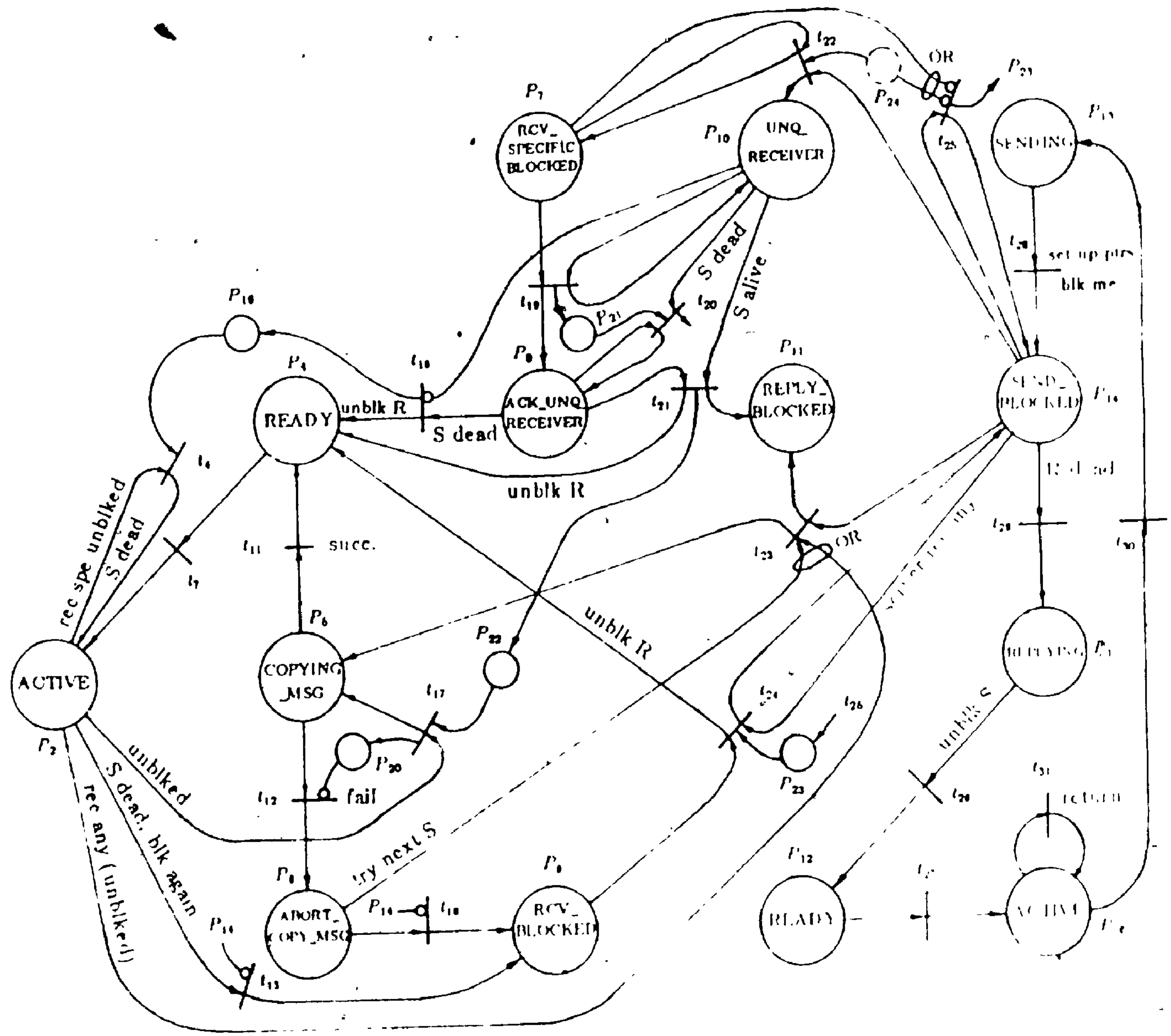


Figure 6.11 _Send()

the receiver. The unblocked receiver fires t_{17} later on, disables t_{12}, copies the message, returns to READY, then ACTIVE successfully.

6.3.5 General Models

Putting above four models together, we get Figure 6.12 and Table 6.4. Basically there is nothing new. So we will not go through it.

An interesting question here is that what kind of relation exists between these two levels of PN models, for example, between Figure 6.1 and Figure 6.12. As we know, the high level model is from the blocking behavior, whereas the low level from the Harmony source code. Hence, it's difficult to find the direct correspondence. The conceptual correspondence does exist. For instance, the sender loop (P_1, t_1, P_2 and t_2) in Figure 6.1 roughly corresponds to all sender's states (P_{16}, P_{13}, P_{10}, P_{11}, P_{14}, P_{15}, and P_{12}) and associated transitions (t_{30}, t_{28}, t_{25}, t_{22}, t_{19}, t_{20}, t_{21}, t_{23}, t_{24}, t_{29}, t_{26}, t_{27} and t_{31}) in Figure 6.11. The clear one-to-one or one-to-a-group corresponding does not exist between these models. However, it's believed that after carefully study, more precise relation can be specified if needed.

To simplify our model and reflect the situation a task faces while it's at ACTIVE state, we merge two pairs of places for states READY, ACTIVE. Moreover, to represent communications between multiple tasks in general, we introduce a new concept—double colored or numbered token to our model.

6.3.6 Double Colored Token PN Models

A double colored (numbered) token is drawn like " ⦶ ". It can represent two tasks by using different colors for or by putting numbers within two half circles. We choose numbers. The left half circle contains ME—the

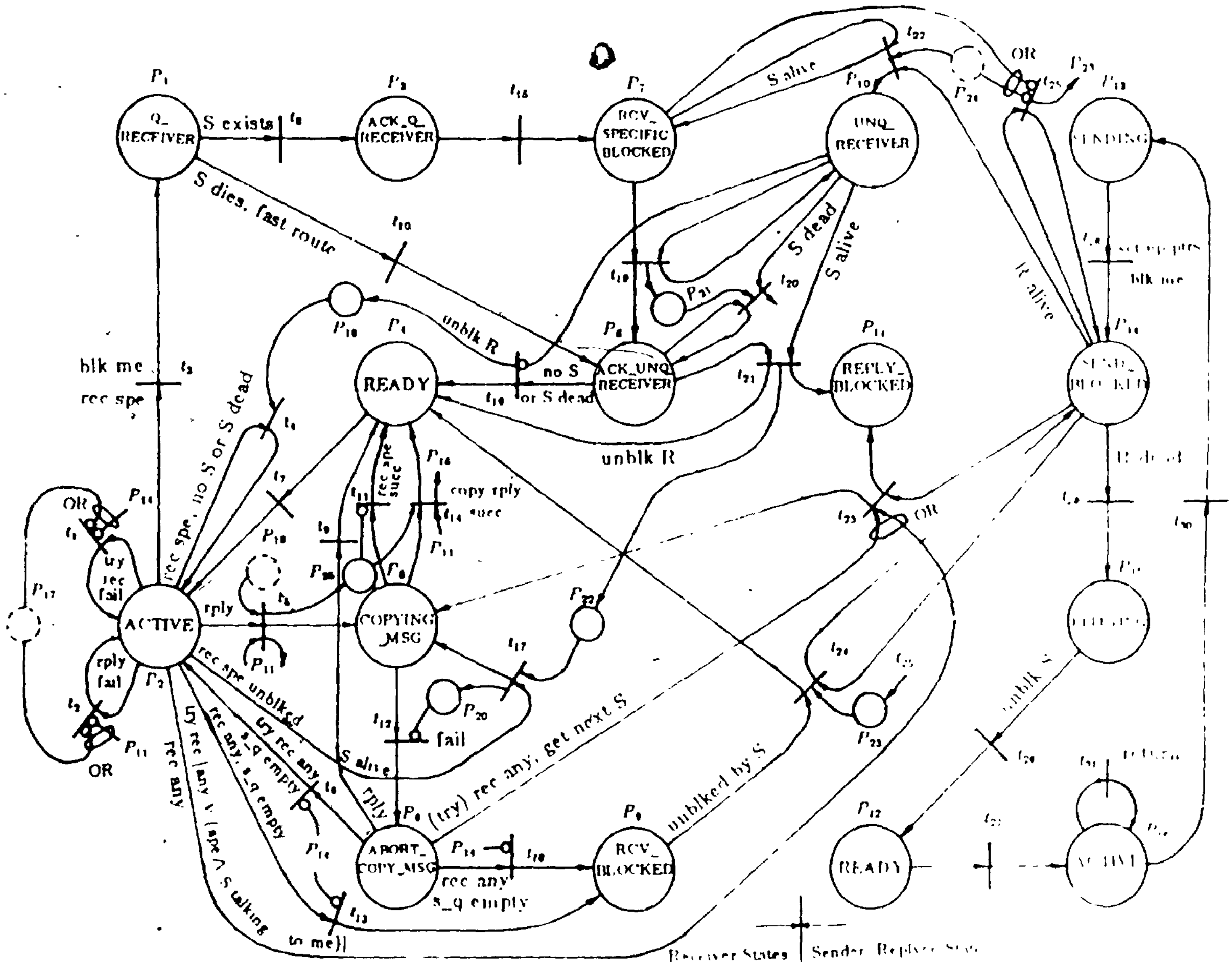


Figure 6.12. A Detailed Overall Model with Dashed Places

caller's id, which can not be zero. If the caller dies, the token disappears. The right half holds my CORRESPONDENT's id, which can be empty in the case of not calling any one of four primitives yet, or 0 in the case of receiving any. Firing rules are revised as following:

a) A transition with one input place fires as usual;

b) A transition with two input places is enabled when a task and its dual (like the image in the mirror) appear in two places. A task and its dual are like:

(4|5) : (5|4) or (4|0) : (N|4) $N \neq 0$

The different cases are shown in Figure 6.13, where t_1, t_2 are enabled, t_3 and t_4 are not. Notice that arcs are also colored.

Now we can examine multiple talks, starting from a easy case—one task tries to receive messages from several correspondents. Figure 6.14 is based on Figure 6.9. From now on the dashed control places are no longer needed.

Task 5 wants to receive a message from task 6, and all messages are available at its send_q. Initially task 5 is in P_4, tasks 6, 7, 8, 9 and 11 in P_{13}. First, task 5 sets its CORRESPONDENT to 6, then enters P_2, calls _Try_receive(). t_{32} is enabled, t_1 not. So t_{32} fires, moves the sender to P_{10} and P_{16}. The caller goes to P_5. Then t_{10} is enabled but t_{17}.

Secondly, task 5 sets CORRESPONDENT to zero, enters P_2 then P_{19}. Tasks 8, 9 are its duals. It removes one, say, task 8 from the head of the send_q (this detail is not represented in our model). t_{26} fires, the caller flows to P_5, enables both t_{10}, t_{17}. If t_{17} fires, the caller moves into P_{19} again. It will remove task 9 and proceed further. If task 5, at this moment, wants to receive

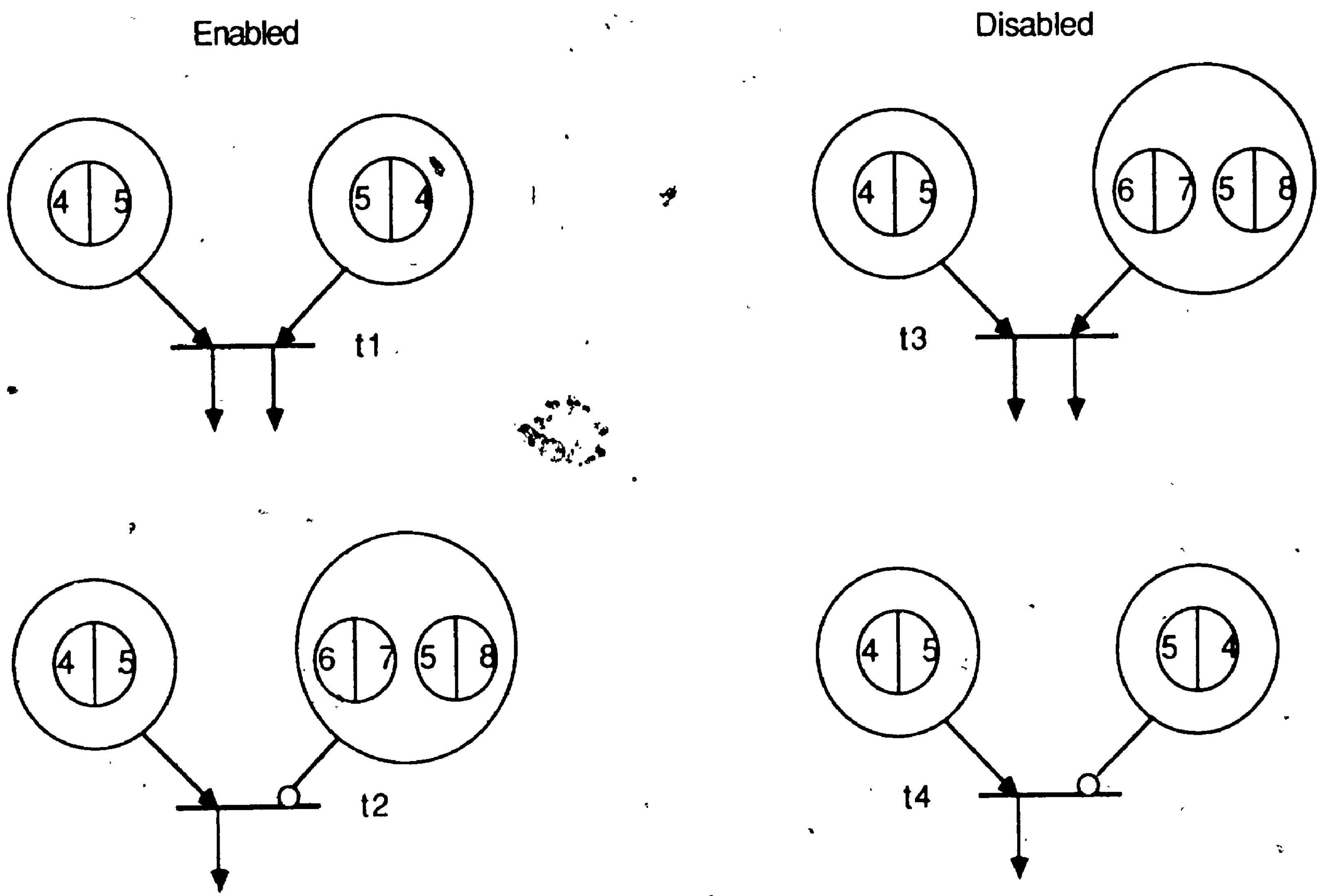


Figure 6.13 Enabled and Disabled Transitions

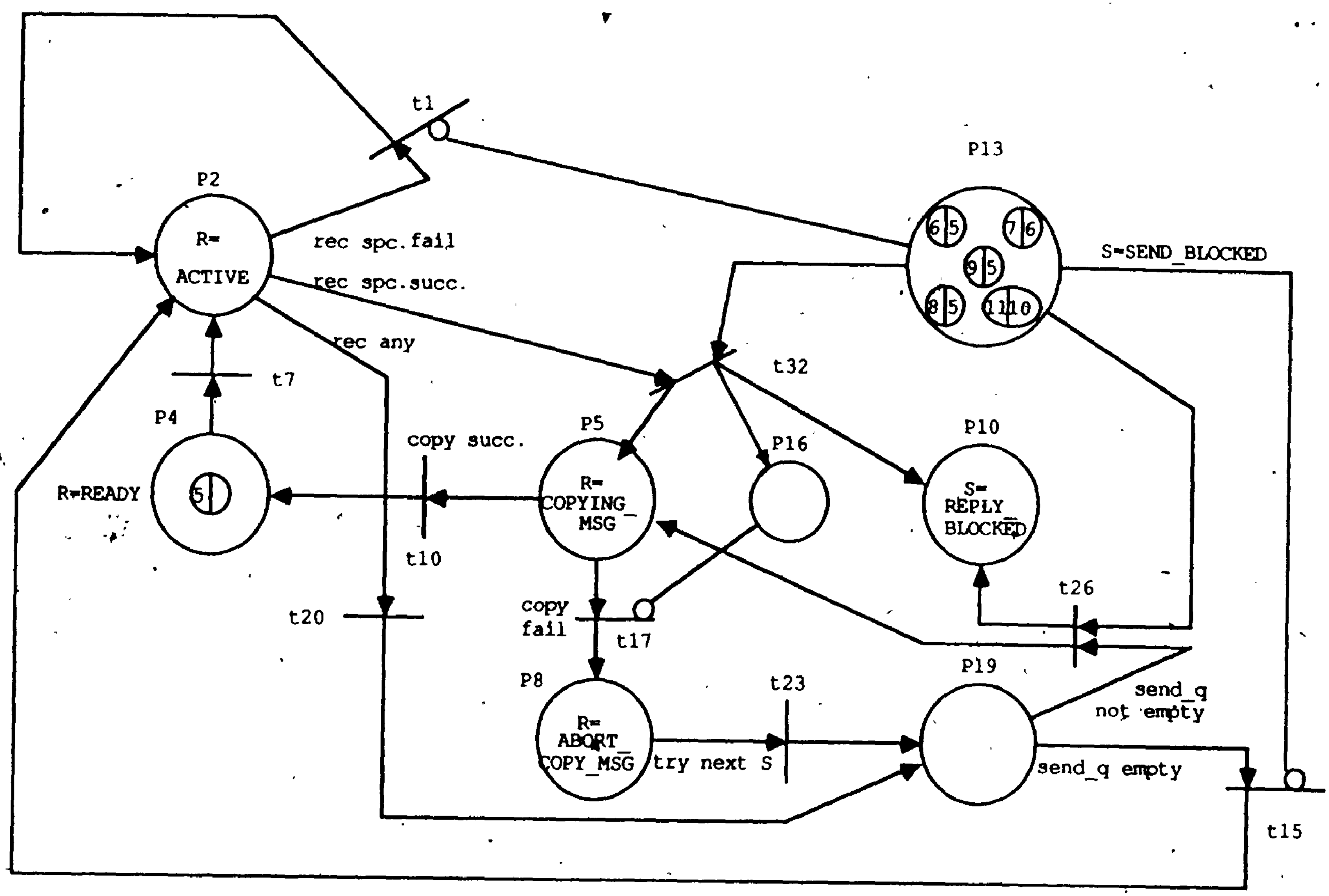


Figure 6.14 Try_receive( )

more messages from its send_q, because two tasks 7, 11 left in P_{13} are not its images (its send_q empty), it fails.

Two things are worth to point out. First, tokens can be accumulated in P_{16}, but it rarely causes trouble. Secondly, when a caller in P_2 intends to receive any, it is smart enough not to fire t_{32} but t_{20}. Modeling all details is possible. The penalty is increased graphical complexity.

The advantages of the double colored tokens are summarized below. First, it removes the dashed places, that solves the problem of how to input tokens into them, and changes the OR-AND transitions to the ordinary ones. All of these simplifies the graphical representations. Secondly, it provides an effective and unique means to check whether a task is its correspondent's correspondent for task communications in general (multiple tasks communicate with each other). For example, in Figure 6.9, if there are five tokens in P_{14} SEND_BLOCKED, then how do we decide the number of tokens in the dashed place P_{17}. Because that number depends on the number of matchings between the senders and receivers. It can be from zero through five. If we set it to five, then what will happen when a receiver out of the matchings (this receiver is not the correspondent task for any senders) comes up. It is only partially correct if the next five receivers fall into the matchings, because by firing t_{32}, the sender removed from P_{14} may not be the exact partner of the receiver removed from P_2.

Nevertheless, when we look at the "receive any", if a receiver is in P_{25}, the five tokens in P_{14} enable t_{23} and it is going to fire. However, probably none of these five tasks might be on the send_q of the receiver in P_{25}, thus t_{23} should be disabled.

In previous studies, the real life (many tasks communicate with each other) was simplified. Take Figure 6.9 for example. In the case of "receive specific", how and when a token is put into P_{17} was not concerned about. In the case of "receive any", the send_q was assumed as the caller's (receiver's) send_q, which implies that all senders possibly in P_{14} are only blocked for the caller instead of for other receivers.

Now we present the final overall PN model in Figure 6.15 and Table 6.5. All kinds of multitask communications can proceed in this model. Compared with Figure 6.12, the two pairs of READY and ACTIVE places are merged. The dashed places are removed, hence the omitted mechanism of inputing tokens to them in previous models is no longer our concern. Subsequently, the OR-AND transitions are reduced to the ordinary transitions.

Some dominating communication rules are inherent in the model:

a) A task can call _Reply() or _Try_receive() as many times and to/from many tasks as it likes;

b) A task can call _Send() only once before being replied, if its correspondent has been alive;

c) A task can call _Receive() to get messages from different tasks as many times as it likes, if each time it receives message successfully.

6.4 Deadlock and its Prevention

Because _Send and _Receive can block, there are some probabilities of deadlock. We discuss it in more details.

Case 1 (sender ring):

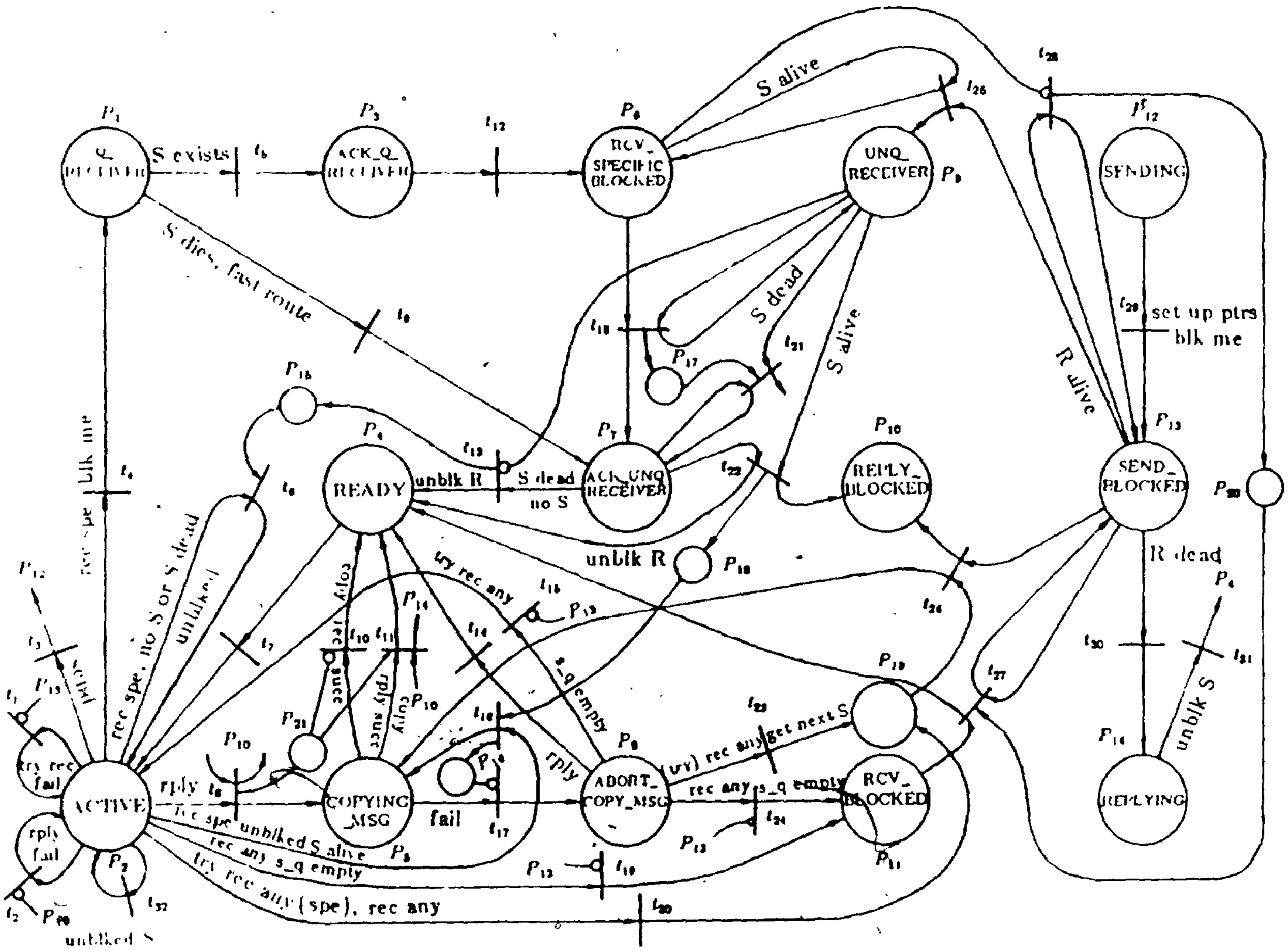


Figure 6.15 A Detailed Overall Model with Double Colored Tokens

Table 6.5 The Second Low Level PN Model

Transition	Equivalent *	Transition	Equivalent
t_1	t_1	t_{15}	t_6
t_2	t_2	t_{16}	t_{17}
t_3	t_{30}	t_{17}	t_{12}
t_4	t_3	t_{18}	t_{19}
t_5	t_8	t_{19}	t_{13}
t_6	t_4	t_{21}	t_{20}
t_7	t_7	t_{22}	t_{21}
t_8	t_5	t_{24}	t_{18}
t_9	t_{10}	t_{27}	t_{24}
t_{10}	t_{11}	t_{29}	t_{28}
t_{11}	t_{14}	t_{30}	t_{29}
t_{12}	t_{15}	t_{31}	t_{26}
t_{13}	t_{16}	t_{32}	t_{31}
t_{14}	t_9		
Place	**Equivalent**	**Place**	**Equivalent**
P_1	P_1	P_{11}	P_9
P_2	P_2, P_{16}	P_{12}	P_{13}
P_3	P_3	P_{13}	P_{14}
P_4	P_4, P_{12}	P_{14}	P_{15}
P_5	P_5	P_{15}	P_{19}
P_6	P_7	P_{16}	P_{20}
P_7	P_8	P_{17}	P_{21}
P_8	P_6	P_{18}	P_{22}
P_9	P_{10}	P_{20}	P_{23}
P_{10}	P_{11}	P_{21}	P_{25}

* The meanings are equivalent to the ones in Table 6.4

Transition	Meaning
t_{20}	calls _Receive any or _Try_receive
t_{23}	tries to get the next sender in the send_q due to failure of copying in _Receive any or _Try_receive any
t_{25}	similar to t_{22}
t_{26}	similar to t_{23}
t_{28}	similar to t_{25}
Place	**Meaning**
P_{19}	for receiver task temporarily

Task A sends a message to B, B sends a message to C, C sends to ...., ... back to A. Each expects its blocked downstream neighbor to release it. But no one can do it. Any task off the ring is not the correspondent of any task in the ring, thus can't help.

For simplicity, we only consider two tasks, say, tasks 6, 7.

In Figure 6.15, task 6 sets its CORRESPONDENT to 7, calls _Send(), enters P_{12} SENDING, then P_{13} SEND_BLOCKED. At this moment, t_{28}, t_{30} are enabled. But t_{30} won't fire, because task 6 "knows" task 7 alive. After t_{28} fires, task 6 will stay in SEND_BLOCKED waiting for its receiver to unblock it. Task 7 repeats the above procedure again, blocks itself in SEND_BLOCK. Each task expects the other to unblock itself, but no one can move. A third task can't help them either, because it's not their correspondent. Deadlock occurs.

In general, the number of blocked tasks can vary from two to a finite large number.

To detect and prevent the deadlock, we can use the following general mechanism :

> Track on the chain of the blocked tasks until the end. Then if the last task is blocked for the caller, that is, the task chain will become a task ring, abort the caller's attempt.

So the cure for the sender ring is as following.

Solution in pseudo code (inserted in the beginning of _Send() primitive) :

```
while( the correspondent is at state SEND_BLOCKED )
 {
   get correspondent's correspondent;
```

```
    if( newly obtained correspondent is the caller )  /* sender ring exists */
      abort _Send();
  }
```

Case 2 (receiver ring):

Task A tries to receive a message from B, B from C, C from A again. All are blocked at the state RCV_SPECIFIC_BLOCKED. In general, the number of blocked tasks runs from two.

Solution in pseudo code (inserted immediately after _Receive() specific) :

```
while( the correspondent is at state RCV_SPECIFIC_BLOCKED )
  {
    get correspondent's correspondent;
    if( newly obtained correspondent is caller )  /* receiver ring exists */
      abort _Receive();
  }
```

Case 3 (mixed ring):

a) Task A sends to B, B receives from C, C sends to D, D receives from A. A and C are blocked at state SEND_BLOCKED, while B and D at RCV_SPECIFIC_BLOCKED.

b) Task A sends a message to task B. B receives it. Instead of calling _Reply(), task B sends a message back to A. Then A blocks in REPLY_BLOCKED, B in SEND_BLOCKED.

In general, the length of task ring could be arbitrarily long. A blocked task can be in one of the three task states : SEND_BLOCKED, REPLY_BLOCKED and RCV_SPECIFIC_BLOCKED.

Solution in pseudo code (properly inserted) :

```
while( the correspondent is at one of three above-mentioned states )
 {
   get correspondent's correspondent;
   if( newly obtained correspondent is the caller )  /* task ring exists */
    {
      if( including caller, more than two tasks on ring )
        abort primitive;
      else if( correspondent at REPLY_BLOCKED ||
                 correspondent at SEND_BLOCKED (caller is a sender) ||
                 corres. at RCV_SPECIFIC_BLOCKED (caller is receiver) )
        abort primitive;
    }
 }
```

Without testing, the revised _Send() and _Receive() primitives are given in Appendix C. The added lines are in boldface. No part of the original code is modified or dropped, though it is necessary to simplify the primitives and make them consistent with the programming style when adding the deadlock prevention algorithm.

The algorithm also can be implemented as a function, in order to hide details from _Send() and _Receive() primitives (see Appendix C).

Chapter 7 Task Creation and Destruction

Task creation and destruction play an important role in Harmony. In this chapter, we present the Petri nets models in both high and low levels. As a review, a brief introduction to the algorithm is given. Then two high level PN models provide us a profile of the algorithm. Finally, the detailed PN models, which precisely summarize and interpret the algorithm, are elaborated.

7.1 Introduction

A program in Harmony is made up of one or several tasks. Tasks are executed in parallel. They communicate and synchronize with each other. Harmony supports multiple tasks on each processor by maintaining a separate priority scheduling system for that processor. Each task is tied to the processor on which it executes, and is added to that ready queue system, as depicted in Figure 7.1 where the contents of the field ENABLE_SR from 0x2100 through 0x2500 are corresponding to processor interrupt levels 1 through 5, by a call to _Add_ready() from the only processor which is the owner of those queues. Level 4 is the highest priority for Harmony tasks. Level 5 is used for _Td_service(). Level 6 is used in _IP_int() partly and in _Disable(). The levels of ready queues can only go up to 5. A task can be created on any processor which might be different from the processor on which the created task is allowed to execute. This adds flexibility for task creation. Similarly, a task can be destroyed on any processor.

Two functions: the _Create() and _Local_task_manager() in Harmony are

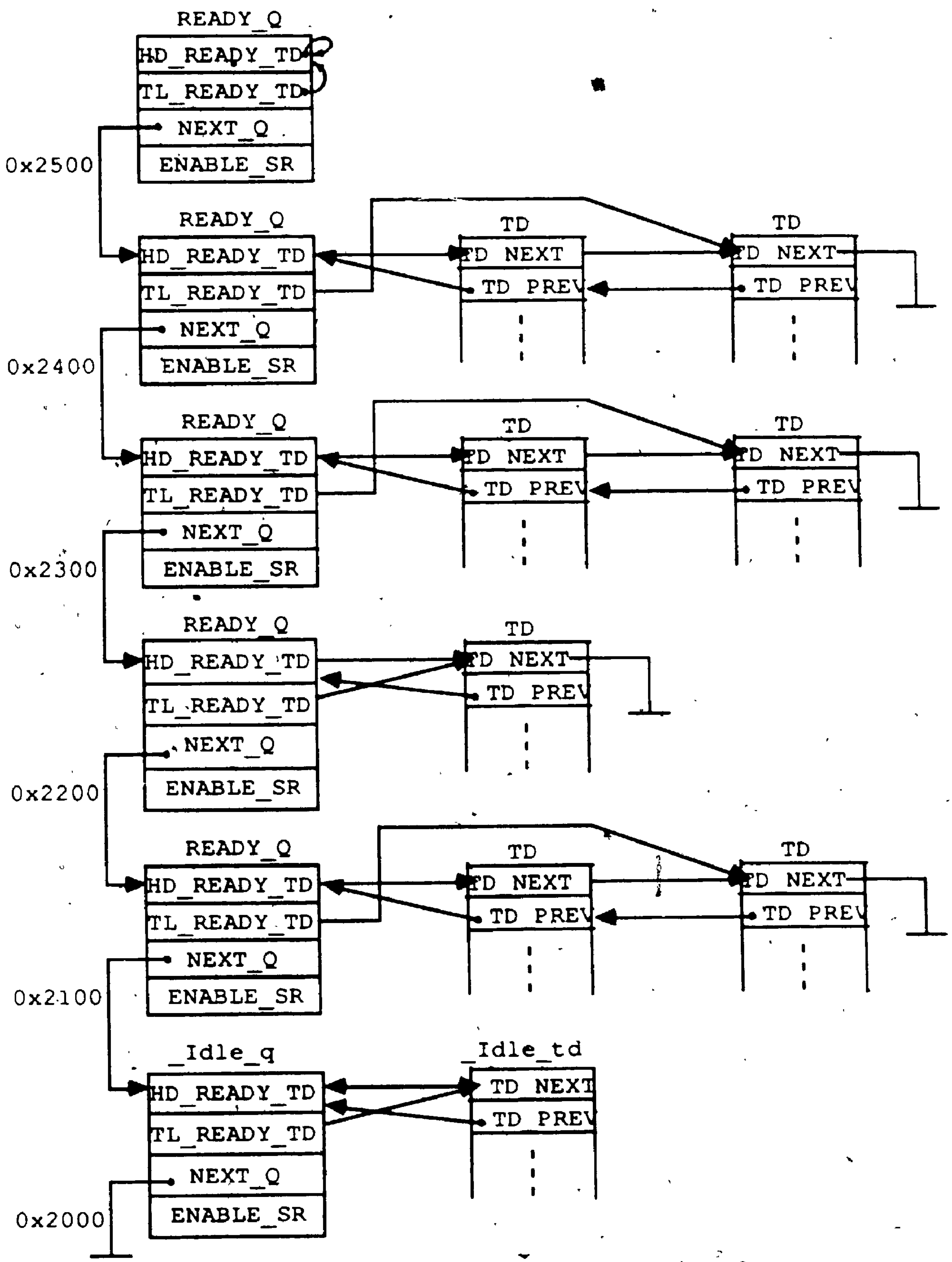


Figure 7.1 Separate Priority Scheduling System

mainly dedicated to task creation, whereas _Destroy(), _Suicide(), _Infanticide() and _Local_task_manager() to task destruction. As a special case, there are three primitives: _I_user_program(), _I_directory() and _I_gossip() responsible for the creation of tasks: user, directory and gossip respectively.

The calling graphs of task creation and task destruction are depicted in Figures 7.2 and 7.3. The wide arrow denotes that the communicating primitives are called and the arrow points to the receiver. During task creation and destruction, some functions will be called but not all of them interest us. Only those requiring maximum synchronizations with other functions are expanded to details in our model. However, for the convenience of reading, we list functions having not appeared before.

_Abort(s) : sends an abortion message to _Gossip() to indicate a fatal error occurred.

_Close(ucb) : a connection no longer needed can be closed by a call to this function. The memory space for the ucb (user connection block) is freed.

_Create(task_index) : creates a task. The task to be created is specified by task index which represents a unique task in the task templates.

_Destroy(id) : the task with task "id" is stopped, all its memory resources are returned, the id is made invalid, all its descendants are killed.

_Free_first_block(td) : frees a block of memory owned by a task with task descriptor pointed to by "td".

_Free_td(td) : frees memory allocation the td points to.

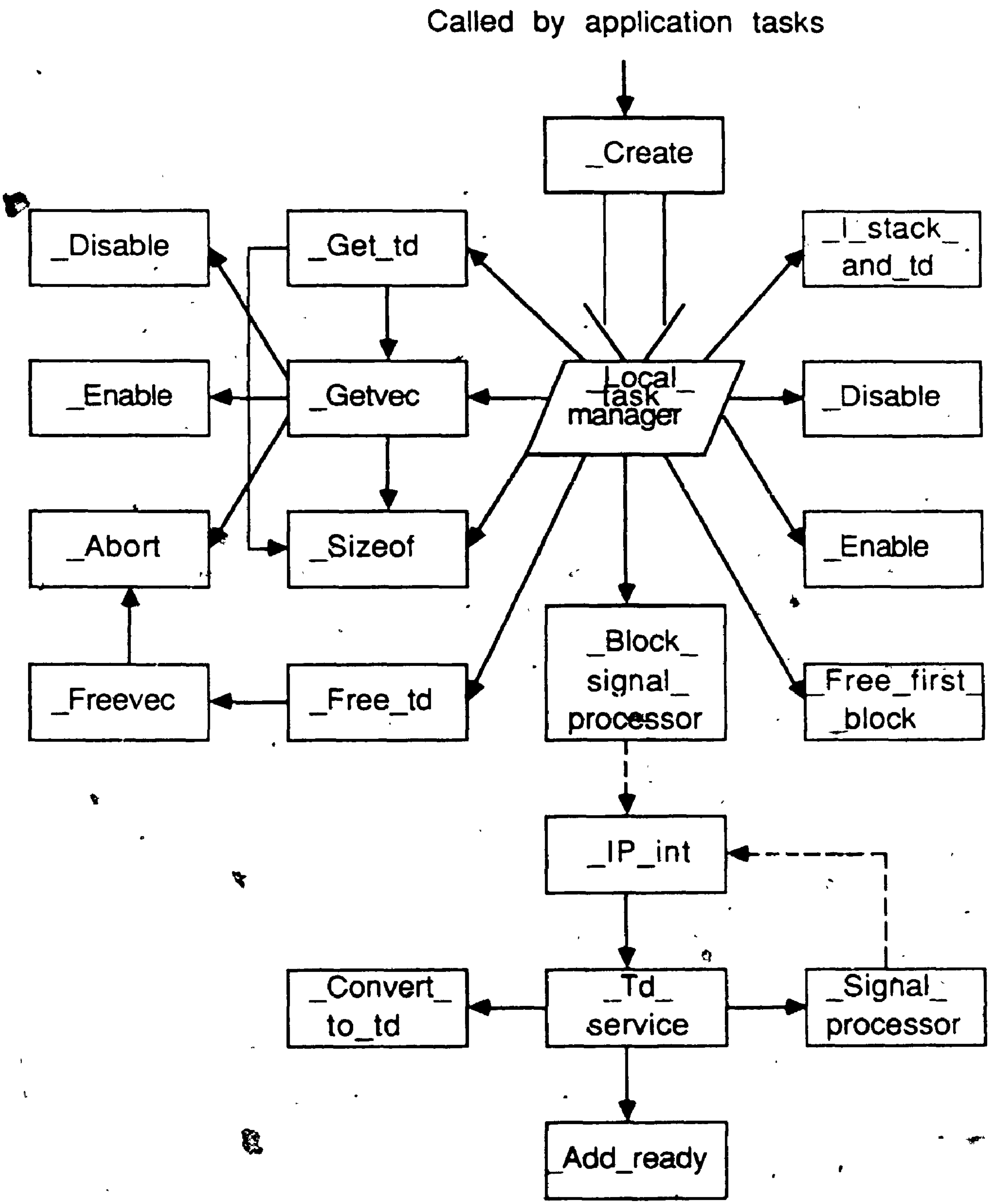


Figure 7.2 Task Creation

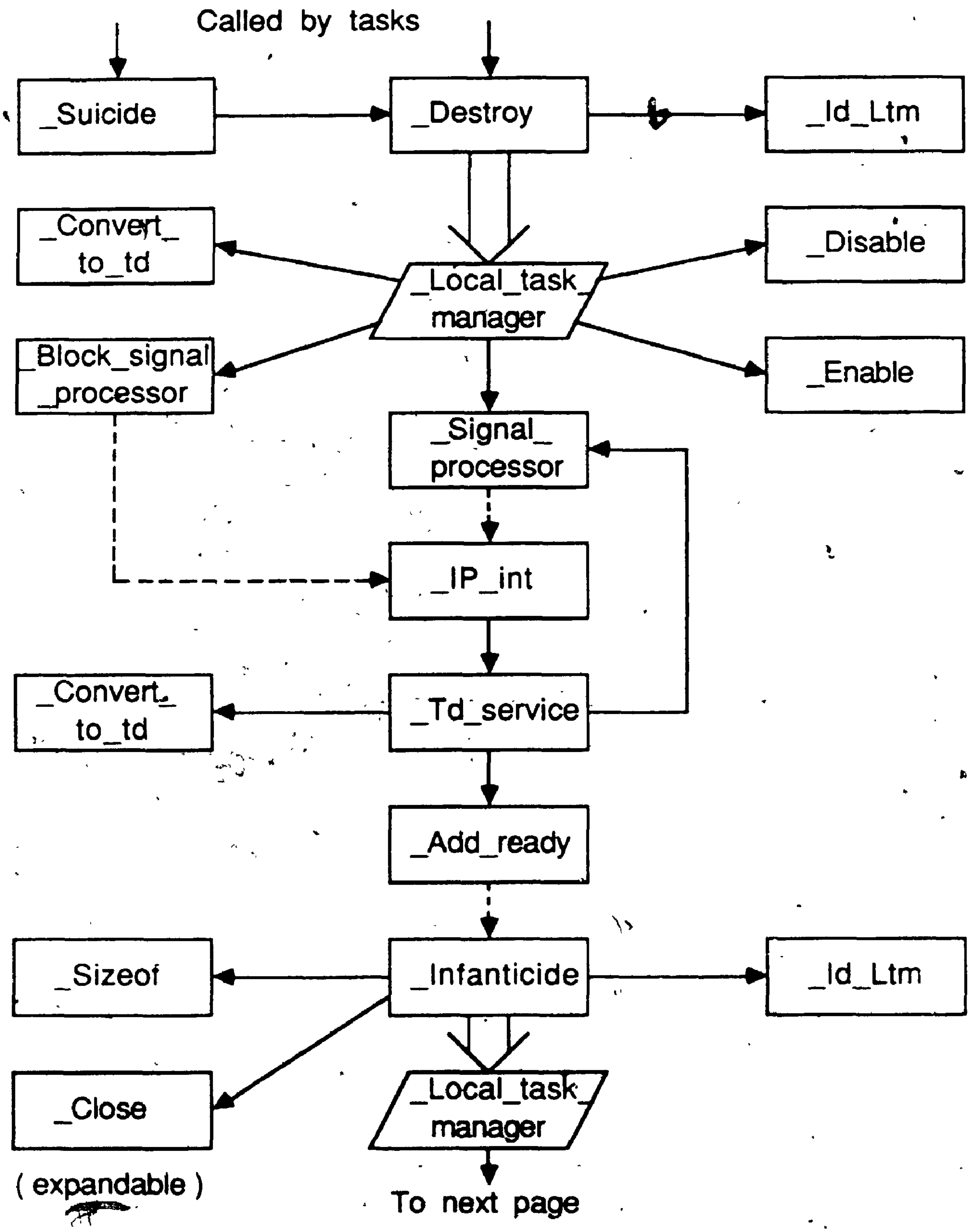


Figure 7.3 Task Destruction (to be continued)

Continued from last page

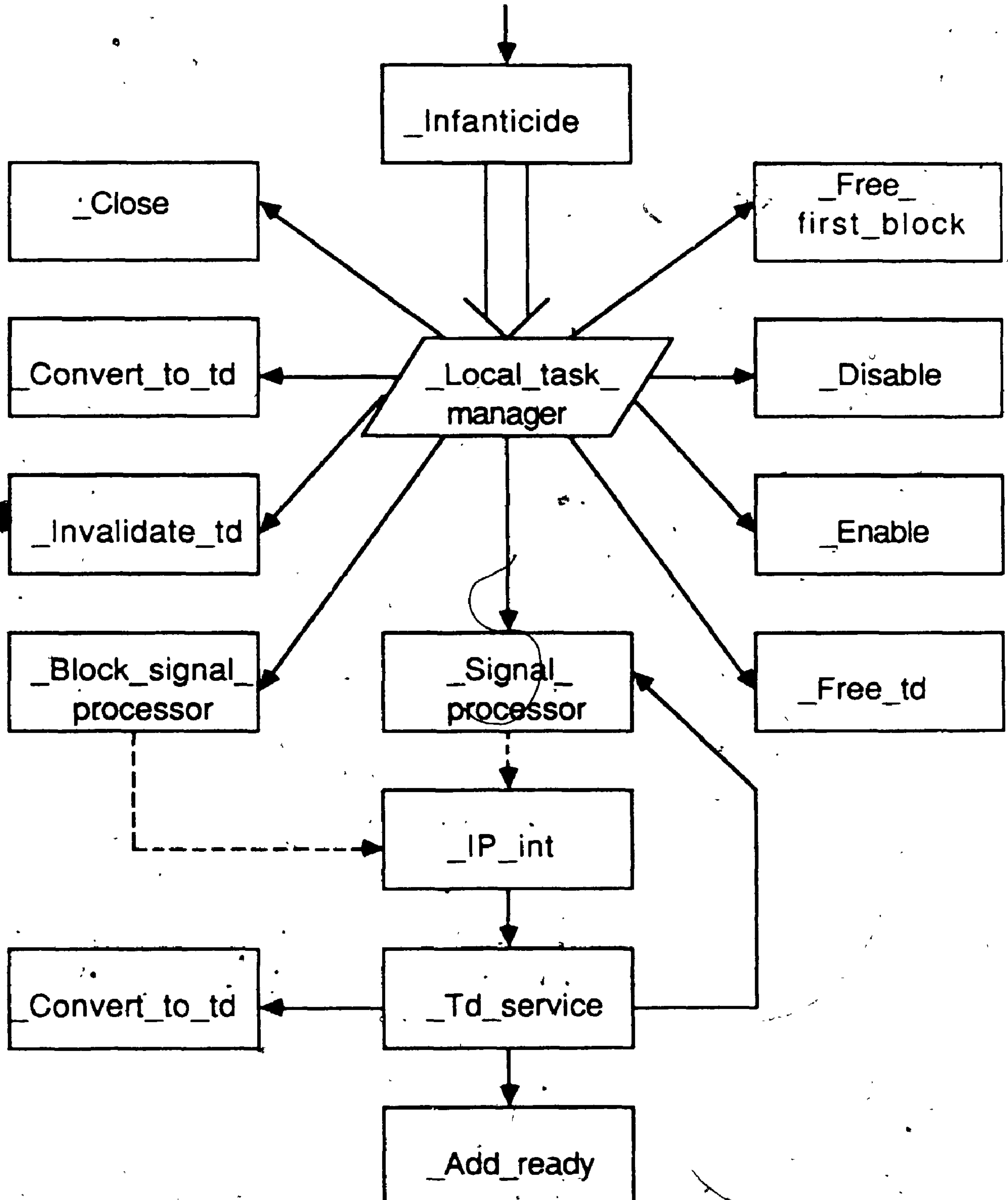


Figure 7.3 Task Destruction

_Freevec(block) : removes the block from the memory resource list for the task.

_Get_td() : gets an empty task descriptor from system, along with a unique task id. Does some initializations. Returns a pointer td to newly acquired task descriptor.

_Getvec(size) : allocates size bytes coalescing on allocation.

_Id_Ltm(id) : returns to the calling task its local task manager id.

_Infanticide(destroyer) : destroys offsprings and closes any connections that the offsprings might have.

_I_stack_and_td(td, stack, stack_start, requestor, root, priority, task_index) : initializes a stack and td for a task by properly setting the fields in these two data structures. Fields like TD_NEXT, TD_PREV, ID, CORRESPONDENT, REQUEST_MSG, REPLY_MSG, LEFT_BROTHER and RIGHT_BROTHER are not filled in by this function.

_Invalidate_td(victim) : invalidates victim's id instead of victim's td by turning off seven bits from the 25th through the 31st. It seems to be more precise if call this function _Invalidate_id(victim).

_Sizeof(block) : returns the size of a dynamically allocated block.

_Suicide() : the calling task destroys itself.

7.2 High Level Petri Nets Models

Similar to top-down design, we start from high level model in order to have a quick understanding. Because our focus is on task creation and destruction now, some less relevant primitives, like _Send(), _Receive() and _Reply() ..., will receive minimum attention.

7.2.1 Task Creation

Each task needs a corresponding task template, which is a data structure specifying the essential parameters of the task [3, p6]. This template can be found in a vector of task template declared for each processor in user's program. A task template is unique in system by assigning a unique integer to one of its fields—GLOBAL_INDEX, and contains other essential parameters like the root function, size of its stack, priority and local task manager which creates and destroys instances of this task.

The data organization for manipulating task templates is depicted in Figure 7.4. The pointer t table is made up of absolute addresses, which provides a matching between the linear addresses and all indices of task templates in system (in Harmony language, the task index, template index and global index are referred to the same thing). The pointer *t table contains the addresses of all task templates in system, provides a matching between absolute address and the address of a task template which requires several memory locations. The **t table provides the memory space for all data structures struct TASK_TEMPLATE where a single slot represents multiple memory locations. Templates in a _Template_list[ ] declared for one processor are in continuous locations from that processor's storage pool.

Next, a task needs a task descriptor, which is a data structure used for an instantiation of a task (template) with a unique id over all task instantiations and stuffed by the system. So multiple instantiations of the task with different

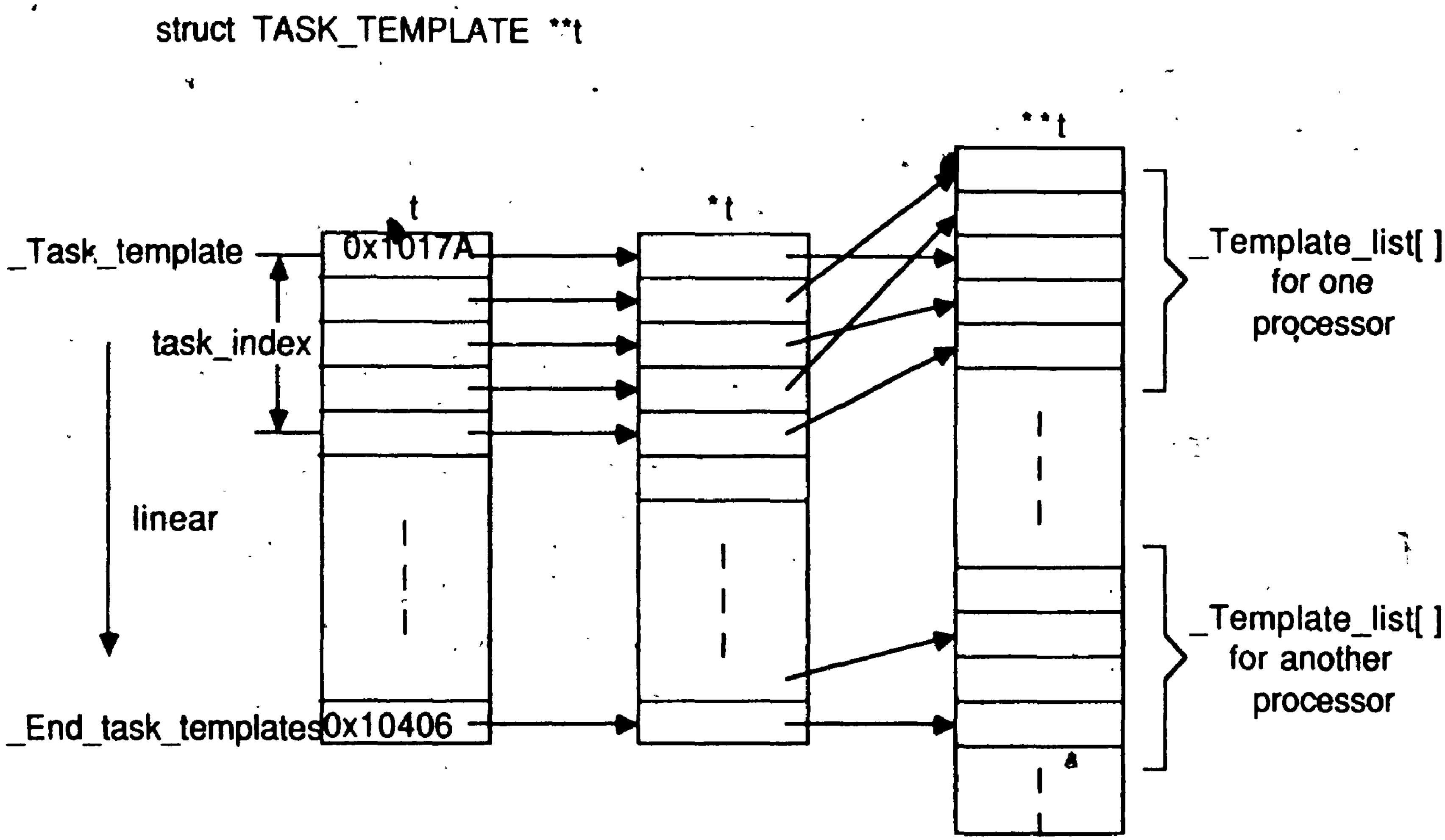


Figure 7.4 Data Organization for Manipulating Task Templates

id can be produced from one task template. Typical fields in a task td are like: state, correspondent, stack, position in a queue, message pointers, its template index, maintained family queue structure, etc.. All these will be properly initialized during task creation.

Finally a newly created task will be put on an appropriate ready queue to wait for dispatch.

Figure 7.5 and Table 7.1 make up our high level model for task creation. A caller starts by calling _Create() from P_1, chooses a task template, then sends a request message to the local task manager, blocks itself in P_2. Upon receiving such a request, the local task manager serves it in cooperation with the primitive _Td_service(). They allocate a td, a stack and other memory resources, then initialize them, add son to creator's offspring structure (queues), finally add both to the ready queue. Eventually t_6 fires. It puts the new baby in P_{10}, releases the creator/father in P_2. The local task manager goes back to P_4, reenters the infinite loop.

7.2.2 Task Destruction

A task can be destroyed by a call to _Destroy() from any other task. It can also commit suicide by calling _Suicide().

Destruction means that the task is stopped, its id made invalid, and all its resources returned to system. Moreover, all its descendants are destroyed. A high level view is depicted in Figure 7.6 and Table 7.2.

_Destroy() starts from P_1. The request goes to _Local_task_manager() (P_3), while the destroyer blocks in P_2. Then the local task manager and _Td_service() serve destruction request, stop the victim, cut off its connection with other tasks, till send the victim to _Infanticide() (P_{10}, t_7).

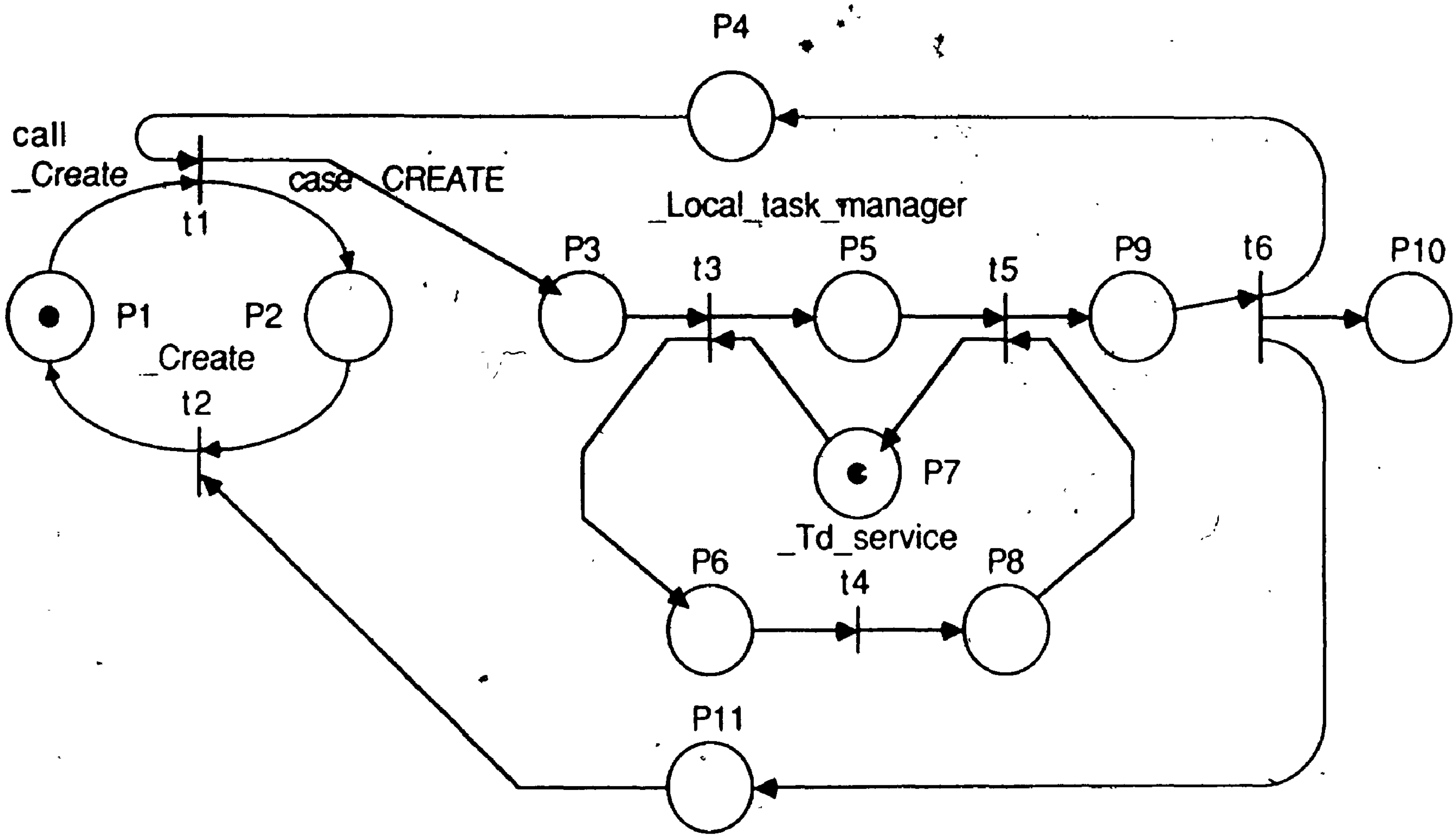


Figure 7.5 Task Creation in High Level

Table 7.1 Task Creation in High Level

Transition	Meaning
t_1	calls _Create(), request msg received by local task manager (ltm), ltm enters case CREATE loop
t_2	_Create() unblocks and returns
t_3	serves request, activates _Td_service()
t_4	has _Td_service
t_5	serves request
t_6	creates new baby, unblock creator, ltm backs to infinite loop
Place	**Meaning**
P_1	creator ready
P_2	creator blocks
P_3	ltm in progress within case CREATE and ready to require td service
P_4	infinite loop entrance for ltm
P_5	ltm waits for completion of td service
P_6	_Td_service activated
P_7	for returned _Td_service, starting place for it, too
P_8	completes td service
P_9	ltm is ready to exit loop
P_{10}	holds new baby
P_{11}	for replied message to unblock the creator

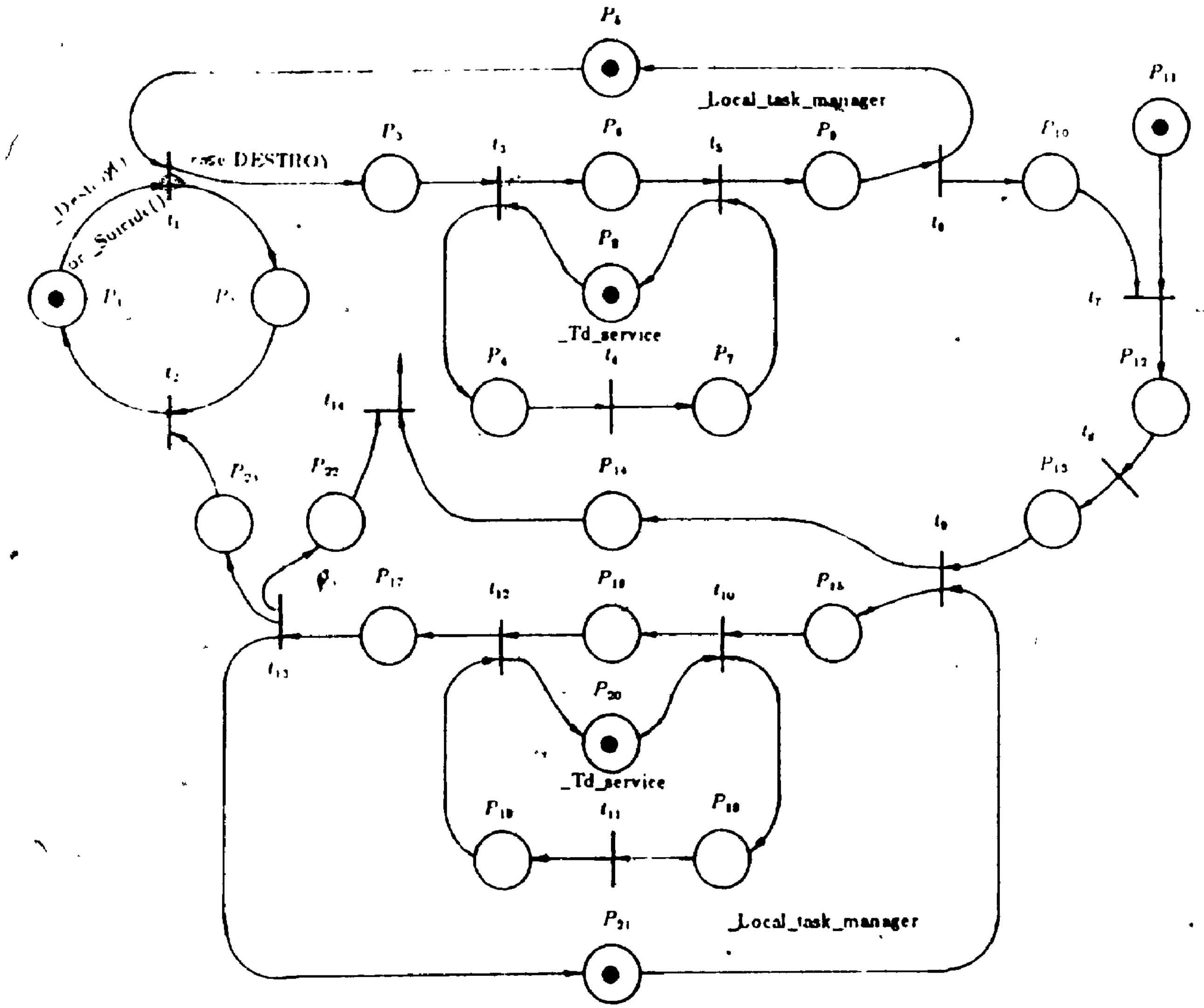


Figure 7.6 Task Destruction in High Level

Table 7.2 Task Destruction in High Level

Transition	Meaning
t_1	sends message to local task manager (ltm), ltm enters case DESTROY loop
t_2	_Destroy() returns
t_3	serves request, activates _Td_service()
t_4	has _Td_service
t_5	serves request
t_6	triggers victim, ltm backs to infinite loop
t_7	activates _Infanticide()
t_8	kills descendants
t_9	activates _Local_task_manager()
t_{10}	ltm serves request, activates _Td_service()
t_{11}	has service
t_{12}	ltm serves request
t_{13}	releases destroyer, ltm backs to infinite loop
t_{14}	wipes victim out of system

Place	Meaning
P_1	destroyer or suicide is ready
P_2	destroyer blocks
P_3	ltm is in progress and ready to request td service
P_4	_Td_service activated
P_5	infinite loop entrance for ltm
P_6	ltm waits for completion of required td service
P_7	completes td service
P_8	for returned _Td_service, starting place for it, too
P_9	ltm is ready to destroy victim's offspring
P_{10}	for a token to trigger activation of victim
P_{11}	holds victim
P_{12}	_Infanticide() activated
P_{13}	ready to kill victim itself
P_{14}	holds victim which is dying
P_{15}	ltm entered case SUICIDE loop
P_{16}	ltm waits for completion of required td service
P_{17}	all destroying works have been done
P_{18}	_Td_service activated
P_{19}	completes td service
P_{20}	for returned _Td_service, starting place for it, too
P_{21}	infinite loop entrance for ltm
P_{22}	for a control token to remove victim
P_{23}	for replied message to unblock destroyer

In _Infanticide(), all victim's offsprings are killed, a suicide request is made to its local task manager by t_9. Activated local task manager puts the victim in P_{14} to let it wait for the completion of its destruction, and as usual cooperates with _Td_service() to serve the request. They remove the victim from its brother queue, release tasks on its message passing queues, free td and memory resources. Finally, t_{13} fires. It releases the destroyer blocked in P_2, drains the victim out of P_{14} through P_{22} and t_{14}. The local task manager backs to the infinite loop (P_{21}) to serve next request.

Notice that when _Suicide() is called from P_1, it resorts to (calls) _Destroy(). The destroyer is now the victim. Having had all services, the local task manager still replies to (by t_{13}) the null victim blocked in P_2 of _Destroy(). Since the victim has gone, _Reply() simply fails. This is represented by the failure branch P_2, t_2 and P_2 again in Figure 6.8. No problem for suicide algorithm.

7.3 Low Level Petri Nets Models

The low level models provide us a close look at the algorithm.

7.3.1 Task Creation

The model is given in Figure 7.7 and Table 7.3. The newly named places are ACTIVE, INFINITE_LOOP_ENTRANCE and CASE_LOOP_ENTRANCE. The partly dotted arc "— ·→" means that some trivials are omitted. For simplicity, we only highlight the improtant aspects of the PN model.

Initially a token—the creator is in P_1, the local task manager represented by a token could be either in P_5 or P_7, _Td_service() is available by a token in

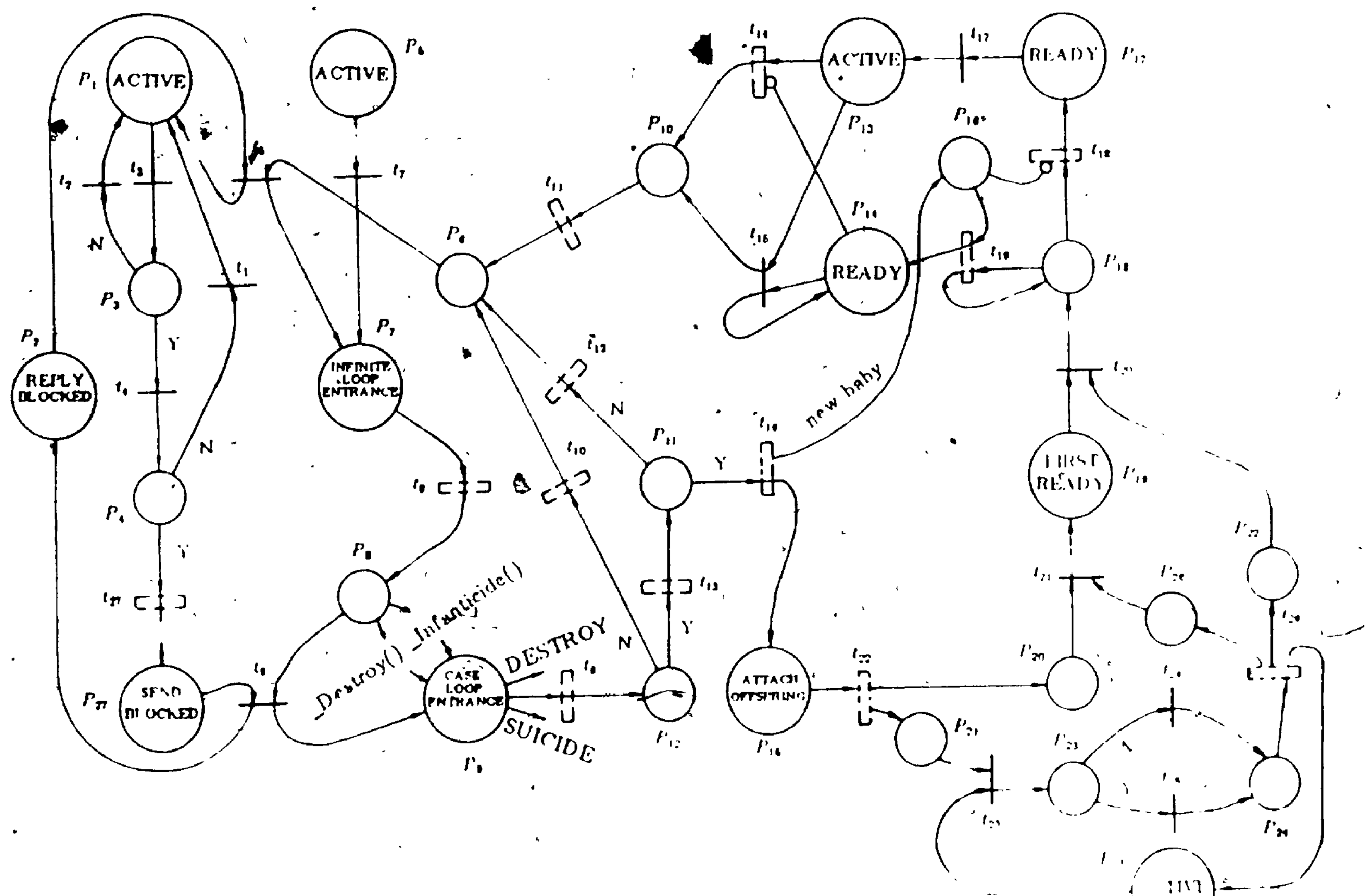
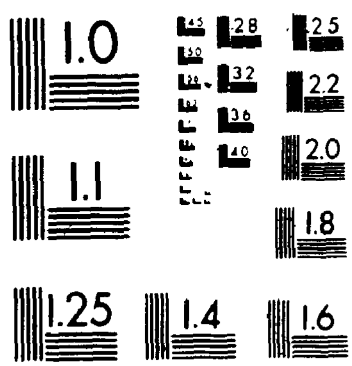


Figure 7.7 Task Creation in Low Level _Create()

2 OF/DE 2



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No 2)

Table 7.3 Task Creation in Low Level

Transition	Meaning
t_1	returns 0
t_2	returns 0
t_3	calls _Create(), checks if the task_index is valid
t_4	checks if the pointer to TASK_TEMPLATE is non null
t_5	local task manager (ltm) unblocks creator
	and reenters the infinite loop
t_6	passes CREATION request message to ltm,
	gets requestor
t_7	ltm activates, sets up response msg pointer
t_8	calls _Receive() any
t_9	gets pointer to TASK_TEMPLATE,
	gets a td and checks if it is good
t_{10}	sets up response msg, calls _Reply()
t_{11}	sets up response msg, replies to and unblock requestor
t_{12}	sets up response msg, frees td, replies to requestor
t_{13}	gets and adjusts stacksize, allocates memory space
	checks if the allocation succeeded
t_{14}	continues execution, frees stack and td
t_{15}	continues execution
t_{16}	initializes stack and td,
	sets ltm's CORRESPONDENT to new baby's id
t_{17}	ltm redispatched (unblocked)
t_{18}	adds ltm to ready queue
t_{19}	adds son to ready queue
t_{20}	ltm enters case FIRST_READY loop in _Td_service
t_{21}	advances ltm's state
t_{22}	blocks ltm, interrupts creator's processor
t_{23}	gets creator's and its son's tds,
	checks if the father is alive
t_{24}	sets ltm's CORRESPONDENT to 0
t_{25}	attaches son to creator's offspring structure
t_{26}	advances ltm's state,
	signals ltm's processor to request service
t_{27}	sets up request msg,
	_Send() to ltm specified in TASK_TEMPLATE

Place	Meaning
P_1	holds the creator—the caller of _Create()
P_2	REPLY_BLOCKED state for the creator
P_3	fork place for result of checking task_index
P_4	fork place for result of checking task template
P_5	holds ltm—the caller of _Local_task_manager()

(To be continued)

Table 7.3 Task Creation in Low Level

Place	Meaning
P_6	ltm ready to reenter the infinite loop entrance
P_7	infinite loop entrance for ltm
P_8	ready to receive a task creation request
P_9	case loop entrance for ltm
P_{10}	ltm ready to reply creator
P_{11}	fork place for result of checking memory allocation
P_{12}	fork place for result of checking td
P_{13}	holds unblocked (redispatched) local task manager
P_{14}	READY state for new baby
P_{15}	ATTACH_OFFSPRING state for ltm
P_{16}	holds new baby
P_{17}	READY state for ltm
P_{18}	ready to add son or ltm to their ready queues
P_{19}	FIRST_READY state for ltm
P_{20}	ltm waits for td service required
P_{21}	for a token to activate _Td-service
P_{22}	for a token to signal ltm's processor
P_{23}	fork place for result of checking the creator
P_{24}	join place
P_{25}	holds caller of _Td_service()
P_{26}	for a token to advance ltm's state
P_{27}	SEND_BLOCKED state for task creator

P_{25}. The creator specifies a template index, calls _Create(), then checks the index, puts the result in P_3 by firing t_3. If a valid task index was chosen, t_4 fires instead of t_2. We can assign two branching probabilities to each branch to simulate token forking. Later the creator sends a request message to the local task manager which is specified in the task template chosen before and is not necessarily the one of creator, then blocks in P_4. If the unborn baby's local task manager has been activated, and is waiting in P_8 (now it's RCV_BLOCKED), then the message will be immediately passed to the local task manager by firing t_6. Otherwise the creator stays in P_4 SEND_BLOCKED. Upon receiving CREATE request, the local task manager will enter case CREATE loop (P_9), then get a td (t_9) and a stack (t_{13}). After initializing them (t_{16}), a new baby is put in P_{16}, while the local task manager enters P_{15} ATTACH_OFFSPRING.

Then the local task manager requests _Td_service, blocks itself in P_{20}. In _Td_service(), the son is added to father's (creator's) offspring structure (t_{25}). The local task manager's state is advanced to P_{19} FIRST_READY. Then if the son is dead (may be killed by another task), the local task manager goes to P_{17} READY. Otherwise the son is first added to son's ready queue by t_{19}, the local task manager follows second from t_{18}.

Now the local task manager may unblock (t_{17}) and continue execution from the old context where it was blocked. It checks the baby's status. If the baby died, it frees stack and td (t_{14}). Finally it replies to the creator (t_{11}), unblocks it (t_5), and reenters the infinite loop P_7.

The conceptual correspondence between two level PN models are given in Table 7.4.

Table 7.4 Correspondence Between Two Level PN Models of Task Creation

Part	High Level (Table 7.1)	Low Level (Table 7.3)
creator	P_1, t_1, P_2, t_2	$P_1, t_3, P_3, t_2, t_4, P_4, t_1, t_{27}, P_{27}, t_6, P_2, t_5$
	t_1	similar to t_6
	t_2	similar to t_5
	P_2	similar to P_2
ltm	$P_4, t_1, P_3, t_3, P_5, t_5, P_9, t_6$	$P_5, t_7, P_7, t_8, P_8, t_6, P_9, t_9, P_{12}, t_{10}, t_{13}, P_{11}, t_{12}, t_{16}, P_{15}, t_{22}, P_{21}, P_{20}, t_{21}, P_{19}, t_{20}, P_{18}, t_{19}, t_{18}, P_{17}, t_{17}, P_{13}, t_{15}, t_{14}, P_{10}, t_{11}, P_6, t_5$
new baby	t_6, P_{10}	$t_{16}, P_{16}, t_{19}, P_{14}$
_Td_service	$P_7, t_3, P_6, t_4, P_8, t_5$	$P_{25}, t_{23}, P_{23}, t_{24}, t_{25}, P_{24}, t_{26}, P_{22}, P_{26}$

7.3.2 Task Destruction

The PN model is given in Figure 7.8 and Table 7.5. A destroyer starts from P_2 ACTIVE by specifying a victim as parameter in _Destroy(). Then it sends a DESTROY request to victim's local task manager by t_2. In the same way as before, the local task manager receives the message by t_4, and enters case DESTROY loop by t_6. It checks the victim's td by t_6. If OK, proceeds and checks the victim's state by t_7. Three possible outcomes are: first, the victim in READY waiting for dispatch, t_{11} fires, removes the victim from ready queue. Second, the victim in AWAIT_INT waiting for an interrupt to signal the completion of I/O service from a device manager, t_{10} fires, removes the victim's td in interrupt table. Third, the victim in one of blocking states like SEND_BLOCKED, REPLY_BLOCKED, ...; and most probably blocking in one of message passing primitives, t_9 fires, the local task manager enters P_{11} RETRIEVING. Then it blocks, sends an interrupt to the victim's processor to request RETRIEVING service (t_{12}). In _Td_service(), if the victim is blocked, removes it from any queue (t_{15}). If the victim's correspondent is copying message, aborts copy message (t_{20}). Then t_{22} fires. It sets the local task manager to ACK_RETRIEVE (P_{22}), and unblocks it (t_{23}). So far the victim is stopped. Next, we will kill it as well as its offspring.

The local task manager will not reenter the infinite loop P_9 until proceeds to t_{26}. There, it sets the victim's state to P_{21} INFANTICIDE by t_{21}, then signals the victim's processor by t_{26}, and the victim joins the ready queue by t_{16} with the entry address of _Infanticide().

The dispatched victim starts from P_{12} ACTIVE, activates _Infanticide(). It closes its connection resources by t_{29}, then checks whether it has sons by t_{31}. If any, it calls _Destroy() to kill it by t_{49}, and checks the next and kills it

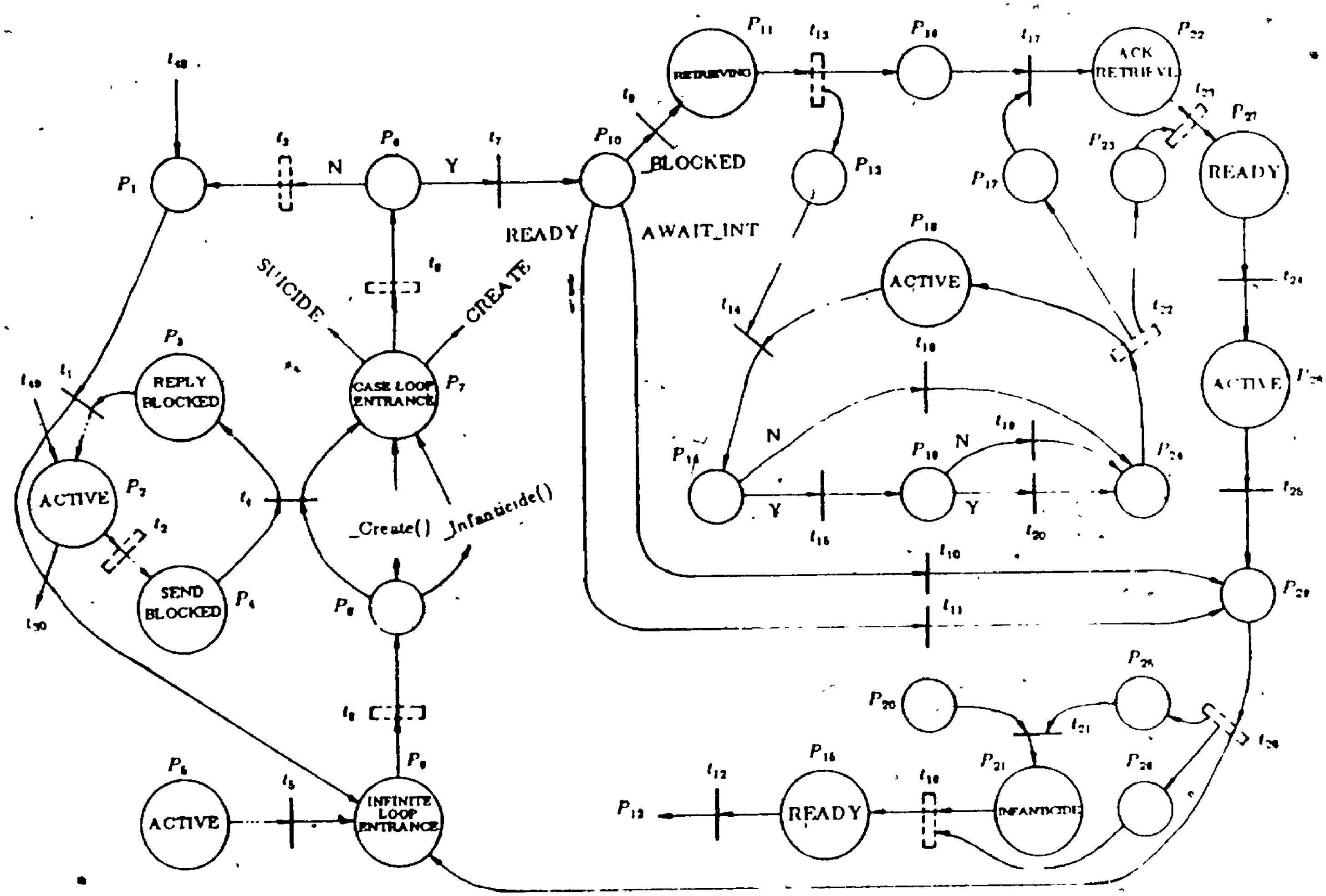


Figure 7.8 Task Destruction in Low Level (to be continued) _Destroy( )

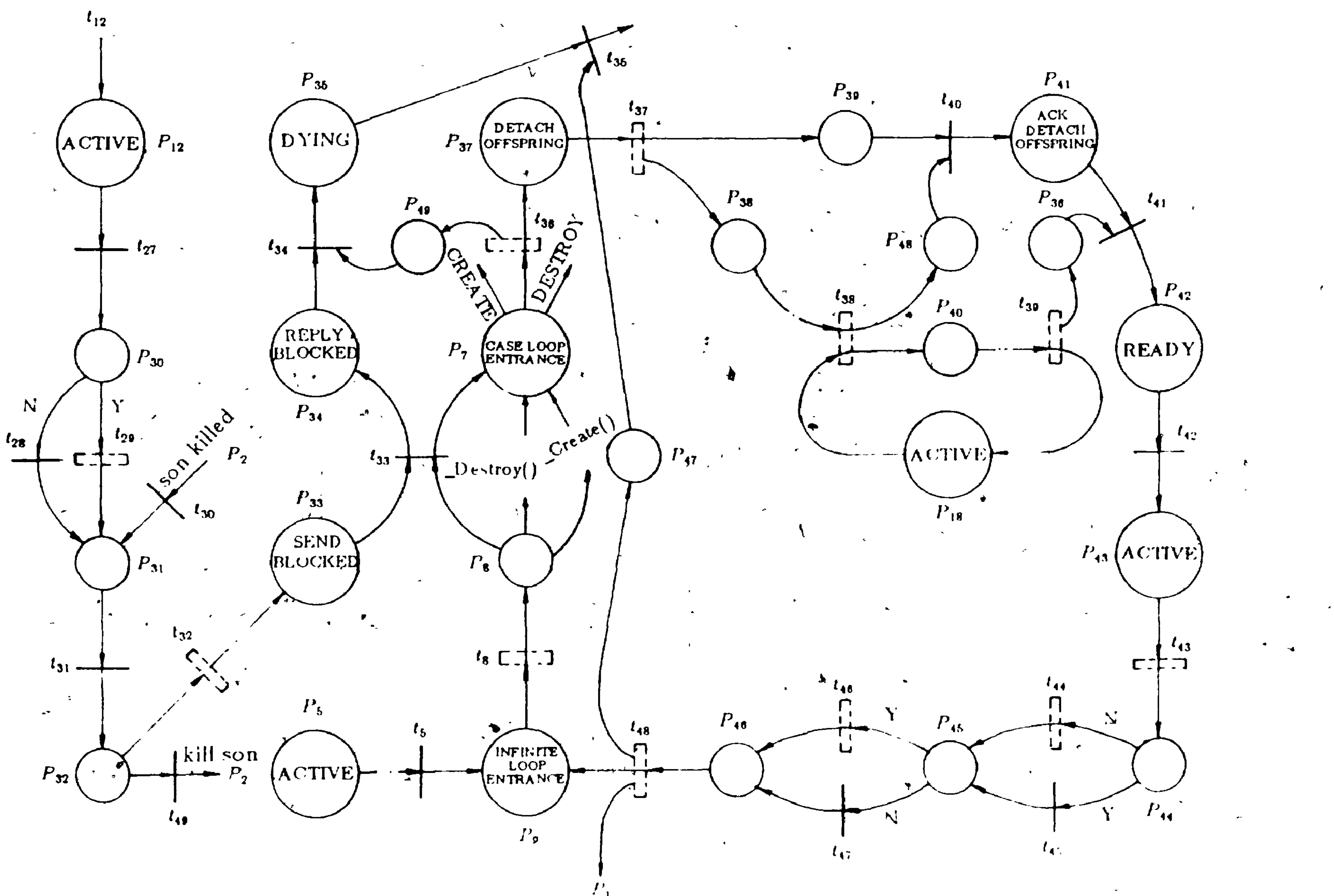


Figure 7.8 Task Destruction in Low Level _Infanticide( )

Table 7.5 Task Destruction in Low Level (To be continued)

Transition	Meaning
t_1	unblocks destroyer
t_2	sets up request msg, calls _Send()
t_3	sets up response msg, replies to destroyer
t_4	passes message to local task manager, gets requestor
t_5	sets up response message pointer
t_6	gets victim's td and checks if it's alive
t_7	checks victim's state
t_8	calls _Receive() any
t_9	sets ltm's CORRESPONDENT to victim's id
t_{10}	removes victim's record in interrupt table
t_{11}	removes victim from ready queue
t_{12}	dispatches the next ready task
t_{13}	local task manager blocks, interrupts victim's processor
t_{14}	get victim's td, check if it's blocked in one of 3 blocking states
t_{15}	removes victim from a queue, gets its correspondent—receiver, sees if receiver is copying message from victim
t_{16}	adds victim to ready queue
t_{17}	advances local task manager's state
t_{20}	aborts copy message
t_{21}	advances victim's state
t_{22}	sets victim's state to RETRIEVED, its CORRESPONDENT to 0, signals local task manager's processor
t_{23}	unblocks local task manager, adds it to ready queue
t_{24}	redispatched
t_{25}	continues execution
t_{26}	changes victim to ltm's priority, assigns victim's STACKBASE, signals victim's processor
t_{27}	has CONN_RESOURCES ?
t_{29}	closes them
t_{30}	returns from _Destroy()
t_{31}	any son exists ?
t_{32}	sets up request message, calls _Send()
t_{33}	passes message to local task manager, gets requestor
t_{34}	advances victim's state
t_{35}	drains victim out of system
t_{36}	gets victim's td, removes victim from ltm's queue, sets local task manager's CORRESPONDENT to victim's id
t_{37}	local task manager blocks, interrupts victim's processor
t_{38}	gets victim's td, removes victim from brother queue, signals local task manager's processor
t_{39}	adds local task manager to ready queue
t_{42}	redispatched
t_{43}	invalidate victim's id,see if its send,recv and reply queues empty
t_{44}	release them, check if victim uses any stack and memory resource
t_{45}	checks if victim uses any stack and memory resource
t_{46}	frees them
t_{48}	releases victim's td, sets up response msg, replies to destroyer
t_{49}	calls _Destroy() to kill son

Table 7.5 Task Destruction in Low Level

Place	Meaning
P	for reply message to release destroyer
P^1	holds destroyer—caller of _Destroy()
P^2	REPLY_BLOCKED state for destroyer
P^3	SEND_BLOCKED state for destroyer
P^4	holds victim's ltm—caller of _Local_task_Manager()
P^5	fork place for result of checking victim's td
P^6	case loop entrance in _Local_task_manager()
P^7	ltm ready to receive a message
P^8	infinite loop entrance for ltm
P^9	fork place for result of checking victim's state
P^{10}	RETRIEVING state for ltm
P^{11}	holds victim—caller of _Infanticide()
P^{12}	for a control token to activate _Td_service()
P^{13}	fork place for result of checking victim's state
P^{14}	READY state for victim
P^{15}	ltm waits for completion of td service required
P^{16}	for a control token to advance ltm's state
P^{17}	holds caller of _Td_service()
P^{18}	fork place for checking result
P^{19}	holds the victim
P^{20}	INFANTICIDE state for ltm
P^{21}	ACK_RETRIEVE state for ltm
P^{22}	for a control token to unblock ltm
P^{23}	join place in _Td_service()
P^{24}	for a token to advance victim's state
P^{25}	for a token to add victim to its ready_q
P^{26}	READY state for ltm
P^{27}	holds unblocked (redispatched) local task manager
P^{28}	join place for ltm
P^{29}	fork place for checking victim's memory resource
P^{30}	join place for victim
P^{31}	fork place for checking victim's sons
P^{32}	SEND_BLOCKED state for victim
P^{33}	REPLY_BLOCKED state for victim
P^{34}	DYING state for victim
P^{35}	for a token to advance ltm's state
P^{36}	DETACH_OFFSPRING state for ltm
P^{37}	for a token to activate _Td_service
P^{38}	waiting place for ltm
P^{39}	required td service has been completed
P^{40}	ACK_DETACH_OFFSPRING state for ltm
P^{41}	READY state for ltm
P^{42}	holds unblocked (redispatched) local task manager
P^{43}	fork place for checking victim's communication queues
P^{44}	transit place for ltm
P^{45}	join place for ltm
P^{46}	for a token to remove victim
P^{47}	for a token to advance ltm's state
P^{48}	for a token to advance victim's state

again, until kills all its sons one by one provided that no grandchildren exist. Notice that this is a self-recursive call. If a tree-like offspring structure exists, it will starts from the bottom—the youngest generation, kills them one by one, then the next old generation, ..., until the second generation. Having killed all its descendants (or no descendants at all), the victim needs to kill itself. It sends a SUICIDE request to its local task manager by t_{32}, later on, blocks in P_{34} REPLY_BLOCKED.

Upon receiving such a request, the local task manager enters case SUICIDE loop by t_{36}, moves the victim to P_{35} DYING, which indicates that the victim is dying gradually, and removes the victim from its reply queue, because the victim is dying, no need to reply/unblock it later. In the meantime, the local task manager enters P_{37} DETACH_OFFSPRING. Having acquired _Td_service, the victim is removed from its brother queue by t_{38}. The redispatched local task manager next checks the victim's send_q, recv_q and reply_q by t_{43}. If finding any tasks there, it releases them and sets the CORRESPONDENT fields in those tasks' td's to 0 to indicate the victim dying (t_{44}). Then it checks the victim's stack and other memory resources, frees them by t_{46}. Next t_{48} fires. It replies to and releases the destroyer blocked in P_3 REPLY_BLOCKED of _Destroy(), drains the victim in P_{35} DYING out of system by t_{35}. The local task manager returns to the infinite loop entrance P_9 to serve the next request.

To commit suicide, a task calls _Suicide(), which in turn calls _Destroy() with suicide's id as the victim.

The correspondence between two level PN models are given in Table 7.6.

Table 7.6 Correspondence between Two Level PN Models of Task Destruction

Part	High Level (Table 7.2)	Low Level (Table 7.5)
destroyer	P_1, t_1, P_2, t_2	P_2, t_2, P_4, t_4, P_3, t_1
case DESTROY of ltm	P_5, t_1, P_3, t_3, P_6, t_5, P_9, t_6	P_5, t_5, P_9, t_8, P_8, t_4, P_7, t_6, P_6, t_3, P_1, t_1, t_7, P_{10}, t_{10}, t_{11}, t_9, P_{11}, t_{13}, P_{13}, P_{16}, t_{17}, P_{22}, t_{23}, P_{27}, t_{24}, P_{28}, t_{25}, P_{29}, t_{26}
instantiation of _Td_service in case DES_ TROY loop	P_8, t_3, P_4, t_4, P_7, t_5	P_{18}, t_{14}, P_{14}, t_{18}, t_{15}, P_{19}, t_{19}, t_{20}, P_{24}, t_{22}, P_{17}, P_{23}
victim	P_{11}, t_7, P_{12}, t_8, P_{13}, t_9, P_{14}, t_{14}	P_{20}, t_{21}, P_{25}, P_{26}, P_{21}, t_{16}, P_{15}, t_{12}, P_{12}, t_{27}, P_{30}, t_{28}, t_{29}, P_{31}, t_{31}, P_{32}, t_{49}, t_{32}, P_{33}, t_{33}, P_{34}, t_{34}, P_{35}, t_{35}
case SUICIDE loop of ltm	P_{21}, t_9, P_{15}, t_{10}, P_{16}, t_{12}, P_{17}, t_{13}	P_5, t_5, P_9, t_8, P_8, t_{33}, P_7, t_{36}, P_{49}, P_{37}, t_{37}, P_{39}, P_{38}, t_{38}, t_{40}, P_{41}, t_{41}, P_{42}, t_{42}, P_{43}, t_{43}, P_{44}, t_{44}, t_{45}, P_{45}, t_{46}, t_{47}, P_{46}, t_{48}, P_{47}
instantiation of _Td_service in case SUICIDE loop	P_{20}, t_{10}, P_{18}, t_{11}, t_{19}, t_{12}	P_{18}, t_{38}, P_{48}, P_{40}, t_{39}, P_{36}

Chapter 8 Error Handling

Similar to the interrupt handling, the processor will enter the exception processing state. But second level handlers are written in C language.

8.1 Description of the Algorithm

Error handling in Harmony is relatively simple compared to the other parts. The whole purpose is to provide a general reporting and logging mechanism to users. Figure 8.1 describes the algorithm employed.

During initialization of processor 0, a call to _I_gossip() from _Local_task_manager creates _Gossip task which reports any errors occurred to the user. Before _Gossip serves any requests, it opens two connections with terminal server by calls to _Open(), then chooses one of them as the input stream and the other as the output stream by two calls to _Selectinput() and _Selectoutput() respectively.

Stream I/O functions: _Putstr(), _Puthex() and _Put() are used to put error messages in output stream. Finally _Flush forces the contents of the output stream out and sends it to the server.

_Gossip receives error handling requests from the interface functions. _Abort() is called from a system function whenever a fatal error occurs in it. The abortion message is sent to _Gossip, then to the user.. In the meantime, _Debug() is activated. The user is able to take proper action, including to shut down the system.

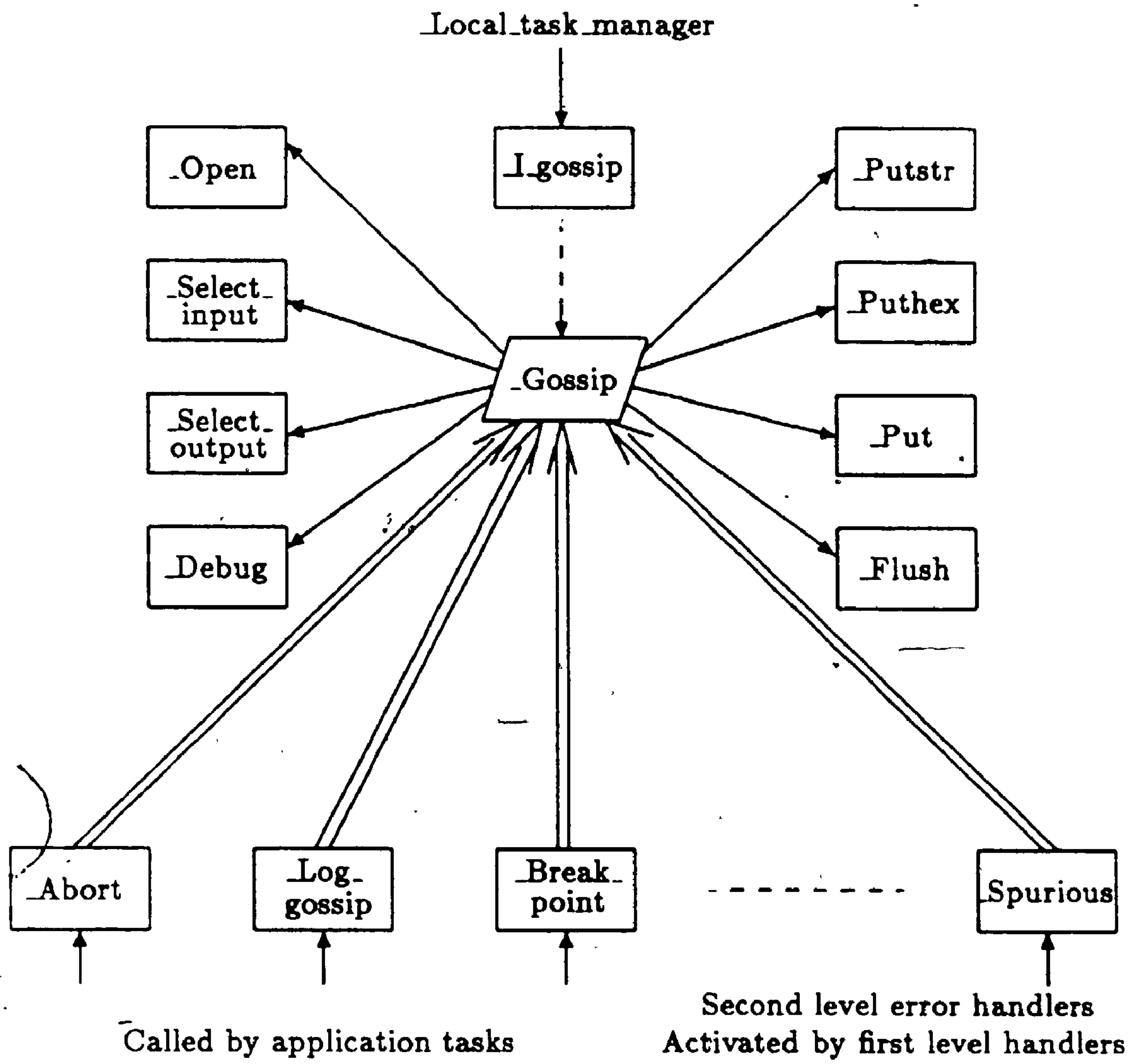


Figure 8.1 Calling Graph of Error Handling

_Debug executes the debugger that provides means of handling errors to user. _Log_gossip is the only interface function which does not invoke the _Debug subsequently.

The MC68000 processor works in one of three processing states: normal, exception, or halted. When some unusual conditions occur and are detected, such as address error, zero divide, privilege violation, ..., as indicated by names of interface functions in Table 8, the processor enters the exception processing state. The current execution is suspended; the context is saved; and a second level error handler is activated which sends a request to _Gossip to display the error message then has _Debug invoked. Having received such a message, say, zero divide, the system user can do something with the help of running _Debug(). After the error has been handled, the previous suspended execution is resumed. The processor returns to the normal processing state.

Harmony kernel also provides three other error handling functions:

_Stackoverflow() : determines if the stack for a task is already overflowed;

_Set_task_error_code(error_code) : sets the task error flag in the task descriptor with the value passed in;

_Task_error_code() : returns the current task's error code.

These three functions reference the task descriptor in the way showed in Figure 8.2.

8.2 Petri Nets Model

The simple algorithm brings us a simple PN model as drawn in Figure 8.3

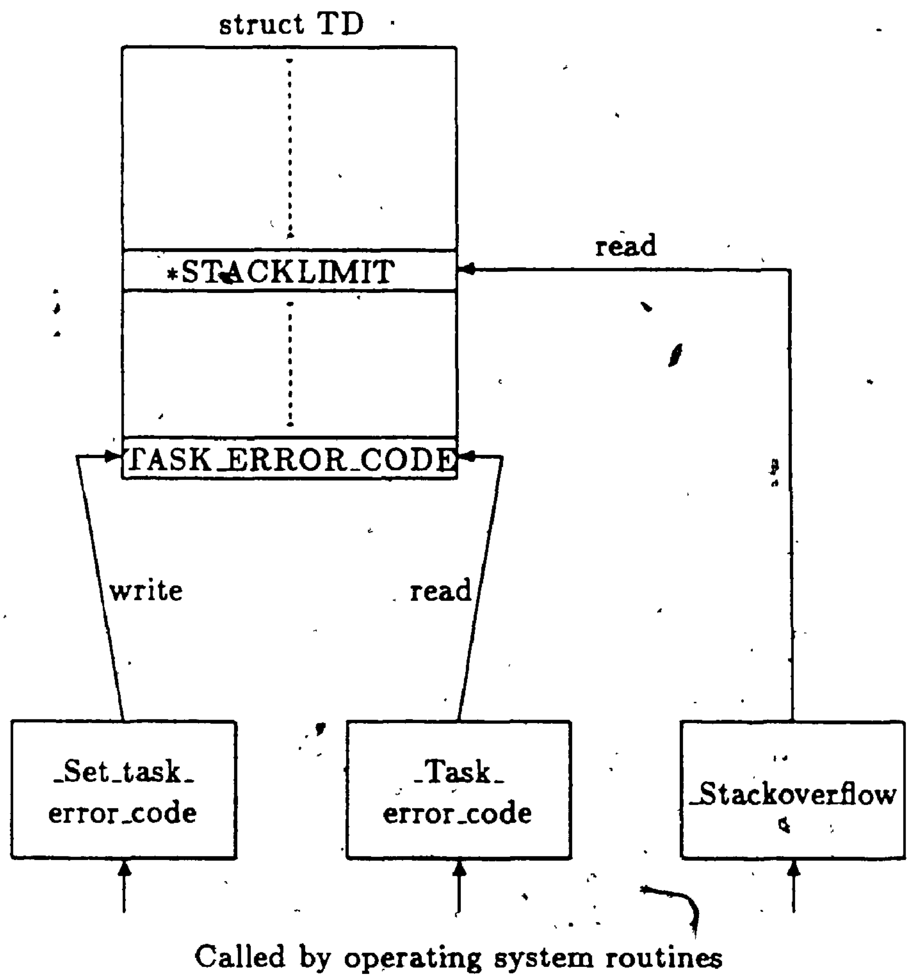


Figure 8.2 Referencing Task Descriptor in Error Handling

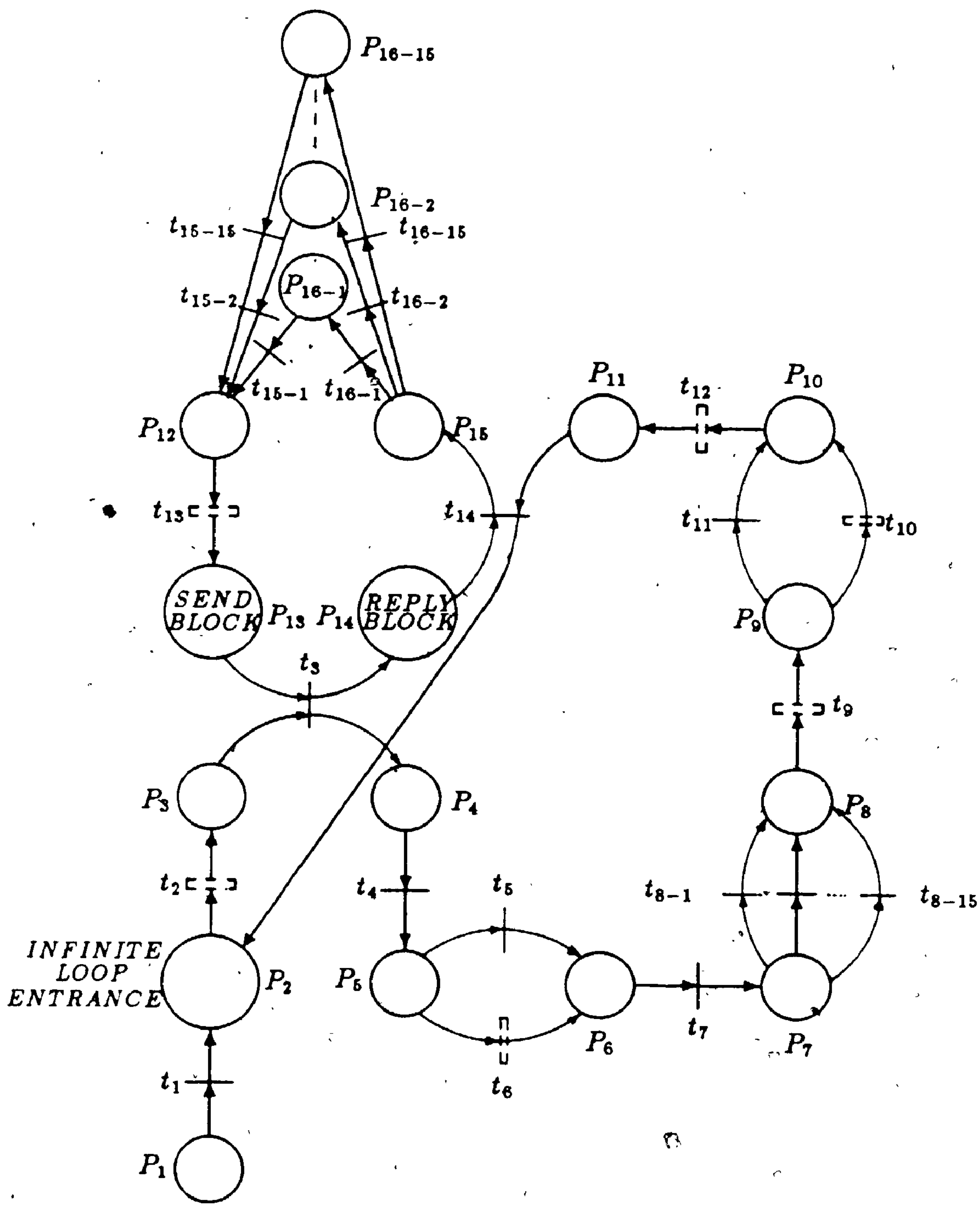


Figure 8.3 Error Handling

Table 8 Error Handling

Transition	Meaning
t_1	invoked, initialized stream pointer
t_2	gets rqst msg size, _Receive() any
t_3	passes message
t_4	is output stream closed?
t_5	dummy transition
t_6	opens and selects connections
t_7	puts requestor id in output stream, checks rqst msg type
t_{8-1}, ..., t_{8-15}	puts error msg in output stream
t_9	outputs the contents of stream,
	is rqst msg type LOG_GOSSIP?
t_{10}	calls _Debug()
t_{11}	dummy transition
t_{12}	_Reply() a null msg to requestor
t_{13}	prepares error handling rqst msg, _Send() it to _Gossip
t_{14}	releases requestor, _Gossip backs to infinite service loop
t_{15-1}, ..., t_{15-15}	error handlers activated
t_{16-1}, ..., t_{16-15}	error handlers return

Place	Meaning
P_1	holds _Gossip
P_2	the infinite loop entrance
P_3	ready to receive a message
P_4	a message received
P_5	fork place for testing of output stream status
P_6	output stream has been checked
P_7	the entrance of switch loop
P_8	the exit of the switch loop
P_9	fork place for testing of rqst msg type
P_{10}	ready to reply
P_{11}	ready to unblock the requestor
P_{12}	the second level error handlers activated
P_{13}	SEND_BLOCKED state for handlers
P_{14}	REPLY_BLOCKED state for handlers
P_{15}	ready to return
$P_{16-1,2,3}$	holds _Breakpoint, _Abort and _Log_gossip
$P_{16-4,5,6}$	holds _Buserr, _Addrerr and _Illinstr
$P_{16-7,8,9}$	holds _Zerodiv, _Chkinstr and _Trapvinstr
$P_{16-10,11,12}$	holds _Privvltn, _Trace and _Em1010
$P_{16-13,14,15}$	holds _Em1111, _Nointvec and _Spurious

and Table 8. _Gossip task starts from P_1. When it proceeds to P_7, there are fifteen output transitions in front of it.

It chooses one according to the message type of error handling request, and usually writes the error message into the output buffer. t_9 tests the request message type. Except LOG_GOSSIP, all other fourteen requests will lead to activation of _Debug().

The places P_{16-1} through P_{16-15} hold fifteen error handlers. The last twelve handlers are second level handlers and activated by the first level error handlers which are hardware implementations. They will stay there before their activations (t_{15}) and after their returns (t_{16}).

Chapter 9

Kernel Supported Server Implementation

In Harmony, a server is referred to as a resource manager responsible for providing services for client tasks. Server implementation is a lengthy part in an operating system. Some common aspects of it are supported by the kernel. In Harmony the kernel is responsible for server creation, initialization and registration, as well as the handling of some common requests cooperating with the servers, such as open and close a connection. The kernel also provides the means of using and monitoring connections.

In this chapter, we give a clear description of the aspects of server implementation which are supported by the Harmony kernel. Section 9.1 provides functional descriptions for each entity and their elements. Section 9.2 is dedicated to the explanations of the algorithms and the dependency among the entities. In Section 9.3, the detailed Petri nets models are presented to describe algorithms precisely.

9.1 Decomposition Description

The software related to this chapter is decomposed into the entities, as depicted in Figure 9.1.

9.1.1 Implementing Servers

Two functions are responsible for server creation, initialization and registration.

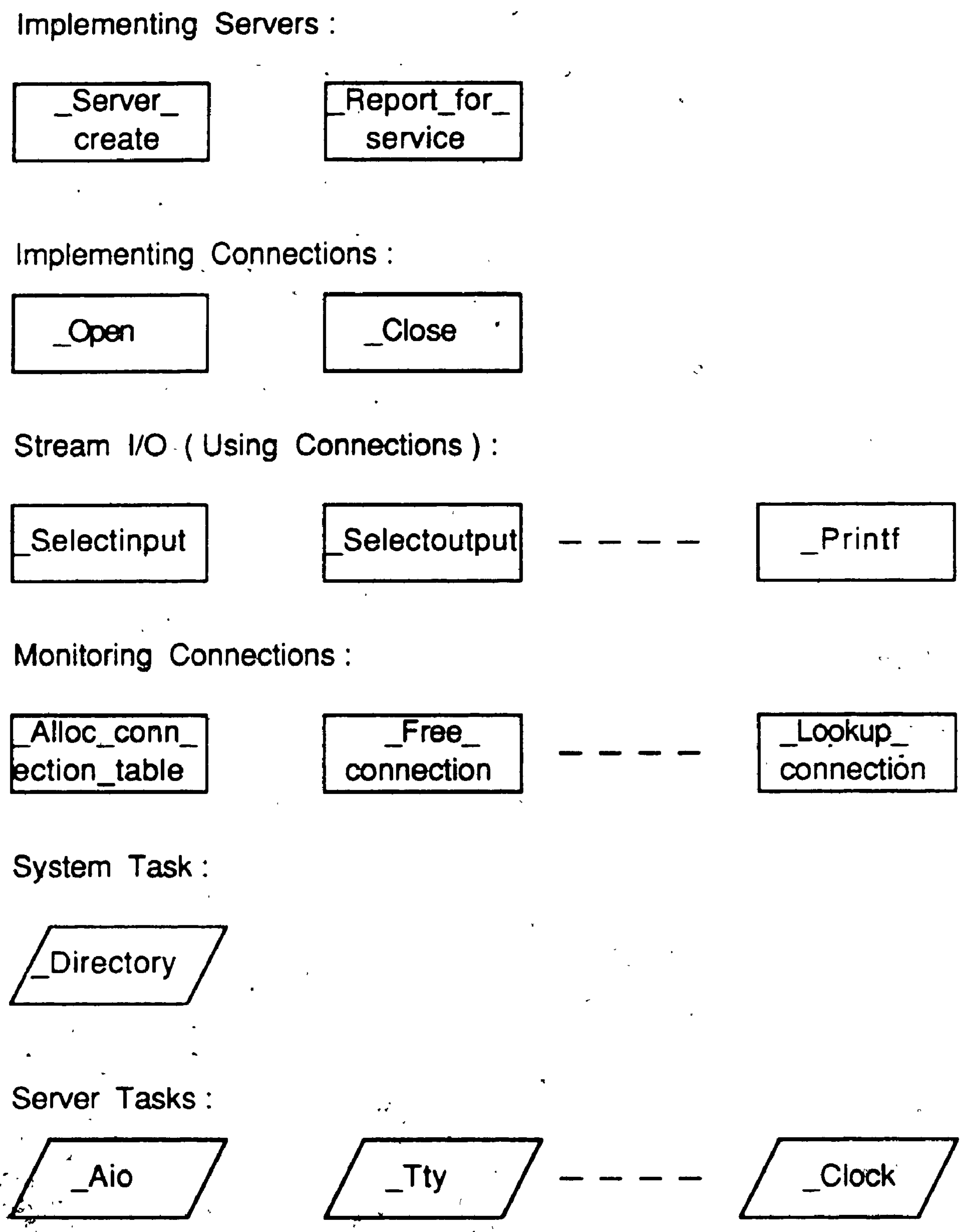


Figure 9.1 Software Classification

_Server_create(task_index, init_list) : called by a user to create a server task needed in his program. The task_index and init_list are prepared by the user for the server to be created.

_Report_for_service(name, msg_type) : called by a server task to register its names and id with _Directory task.

9.1.2 Implementing Connections

Two functions in this entity are called by a client who wants to do I/O with an I/O server. A connection, which is in fact a shared buffer between a client and a server, is the means for doing such I/O.

_Open(name, mode, user_id) : a connection can be opened by a call to this function with the name of the server and the mode to use this connection specified in argument list.

_Close(ucb) : a connection no longer needed can be closed by a call to this function. The memory space for the ucb (user connection block) is freed.

9.1.3 Stream I/O

A stream is an infinite sequence of bytes, a connection/buffer between a client and a server. This entity provides the means for user in correctly using such a stream.

_Selectinput(ucb) : called by a client to select one of the opened streams for input.

_Selectoutput(ucb) : called by a client to select one of the opened streams for output.

_Flush() : called by a client to output the contents of buffer to the server, no matter whether the output buffer is full or not.

Descriptions of other functions in this entity can be found in the Harmony user manual, hence we do not list them here.

9.1.4 Monitoring Connections

A server uses functions below to keep track of those connections it has opened with clients.

_Alloc_connection_table(init_num_entries, scb_size) : the connection table is used to hold the records of opened connections. A server calls this routine to allocate a connection table.

_Get_connection(table, client, new_connection) : a server calls it to allocate an entry in a connection table.

_Lookup_connection(table, client, connection) : called by a server to look up an entry from the connection table. It verifies the client and the connection.

_Grow_connection_table(table) : grows a connection table by the CON_GROW_AMOUNT specified in the "table". It is called by _Get_connection when necessary, never by a server.

_Free_connection(table, connection) : called by a server to free a connection in a connection table.

9.1.5 System Task

Only one system task is in this entity.

_Directory() : it is the root function for the directory task, which provides server task's id when server's symbolic name is submitted by a client.

9.1.6 Server Tasks

Most servers are either I/O servers or have I/O features, such as the clock server. A server is defined either by system designer or user based on the peripheral it controls and service it provides. Consequently, their structures vary greatly from a few lines of task, such as _Explicit_Scheduler, to a hierarchical task and routine family, such as _Clock_server. The detailed classification and discussion are beyond the scope of this chapter since they are device/model dependent. We only list two system defined servers as they are needed shortly to clarify our study.

_Aio_server() : is the root function of the analog I/O server. This server provides both A to D and D to A, which are really the details we are not interested in here.

This server does not have any work/agent tasks, nor interface routines, but is a simple "bachelor". As we will see later, it requires only one initialization record.

_Tty_server() : is the root function for terminal server, and the administrator for _Tty model. It has a _Tti handler and a _Tto handler. All initialization works are moved to _I_tty_server(tty_state), which creates the _Tti and _Tto tasks and initializes them as well as the tty_state record.

9.2 Algorithm and Dependency Descriptions

In Harmony, servers can be defined either by system designer or system user. It is user's responsibility to create whatever servers he needs in his program. Upon creating a server, the kernel will look after server initialization and registration with _Directory task. Then a client task can request service by opening a connection with this server to do I/O. After having service acquired, the client may close the connection. A client can request service through interface to the server, as well.

9.2.1 Server Creation, Initialization and Registration

To create a server, an initialized task template is added onto the _Template_list[ ] at the user's program. The user also needs to prepare a list of initialization records for the server. They will be replied to the server one at a time. The size and format of the initialization records are server dependent, but the struct INIT_REC must be at the start of each record.

Such a list of records are depicted in Figure 9.2. The list head pointer is set in _Server_create(). MSG_SIZE is the size of whole record. MSG_TYPE is prepared for a server to distinguish the various types of initialization records. IR_NEXT field is for the address of the next initialization record. The record with server's primary name is put at the start of the list. For the _Tty_server there are two records, while for the _Aio_server only one record. If no initialization is needed for a server, the list is empty.

The server creation, initialization and registration are depicted in Figure 9.3.

From the user task's root function main(), the function _Server_create() is called with the specified parameters task_index and init_list which were prepared by the user. _Server_create calls _Create() to generate the server task. The created server then sends an initialization request to _Server_create which in turn replies one initialization record at the head of the init_list to that server. The server checks the type of the replied initialization record. In

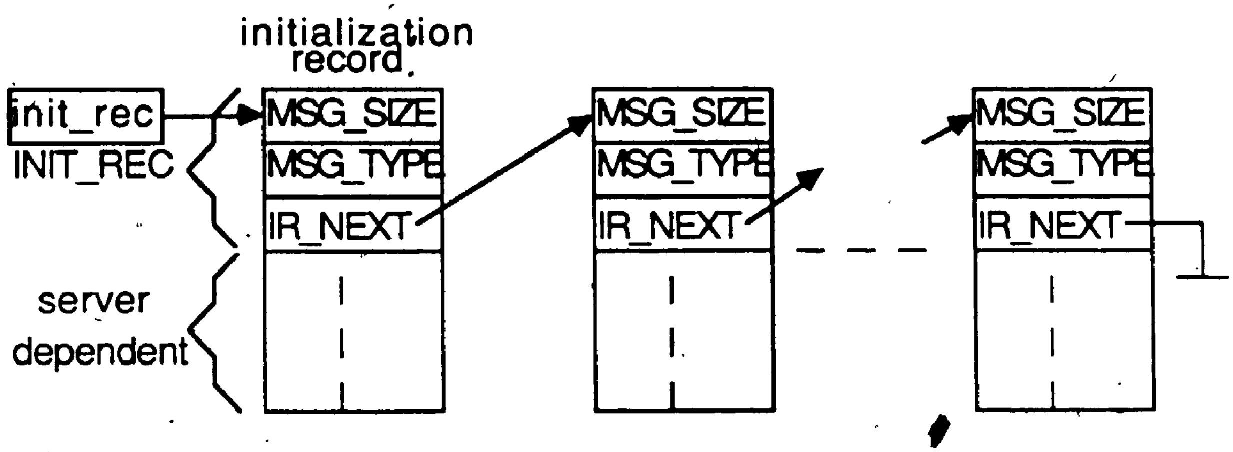


Figure 9.2 A List of Initialization Records for a Server

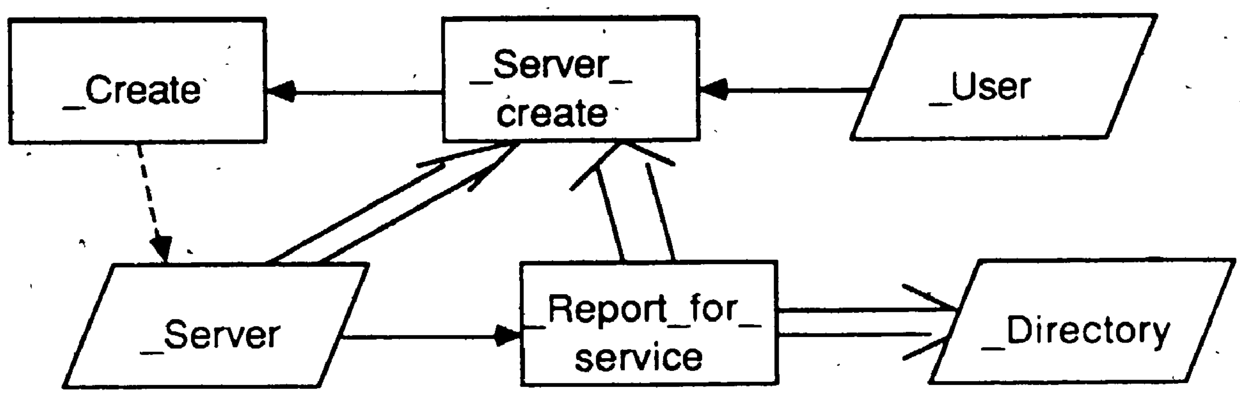


Figure 9.3 Server Creation, Initialization and Registration

general, there can be three cases to deal with.

First, a null record is found, i.e., the server does not need initialization.

Second, only one record is on the init_list, such as for _Aio_server. The server next calls _Report_for_service() to do REPORT_FOR_SERVICE, that actually sends the server name to _Directory task.

_Directory maintains two server name lists: the server_list and the secondary_list. The server_list is for all kinds of servers' names, whereas the secondary_list is only temporarily for server's secondary names. Their structures are shown in Figure 9.4. Both lists are made up of SERVER_ENTRYs. Later on a client will provide a server name to find the server id. The "list_ptr" only points to one of lists at any time.

After received the server's name for _Aio_server, _Directory checks its validity, allocates a SERVER_ENTRY, writes the name and id into the entry, and adds that entry to the head of the server_list. Then _Directory replies a registration OK message to the server in _Report_for_service(), where the server in turn replies a REPORT_COMPLETED message to its father in _Server_create(). Then the server enters the infinite loop to provide service for clients, whereas its father exits _Server_create() and proceeds further.

Third, there are more than one records on the init_list, such as the two for _Tty_server. Dr. Gentleman's algorithm requires that any secondary names must be reported to _Directory prior to the REPORT_FOR_SERVICE request, although the first received record contains server's primary name. The server merely takes down such information, probably does something else next, then it checks whether there are more initialization records on the

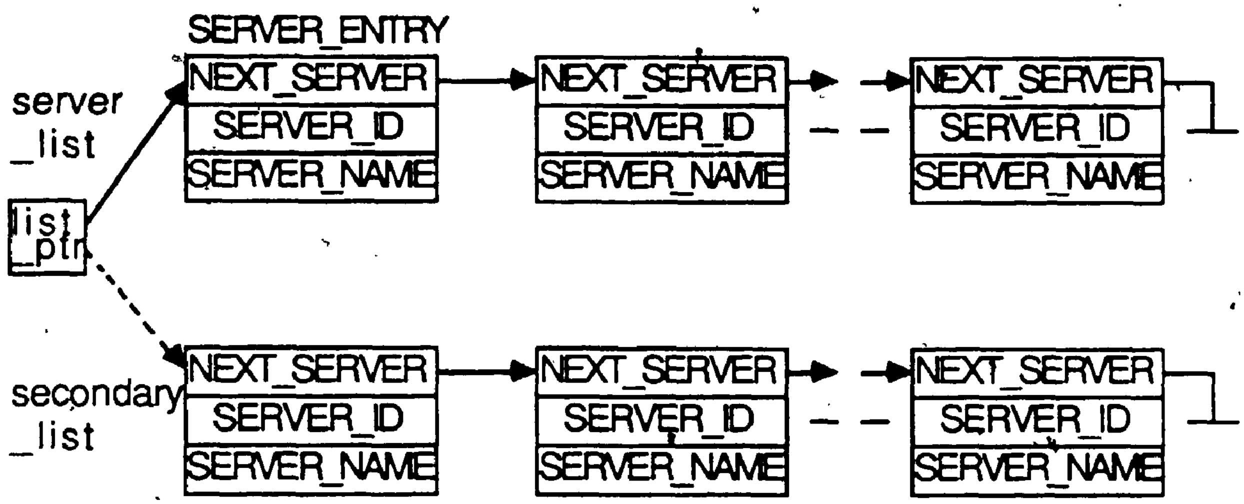


Figure 9.4 Server List Manipulated by _Directory

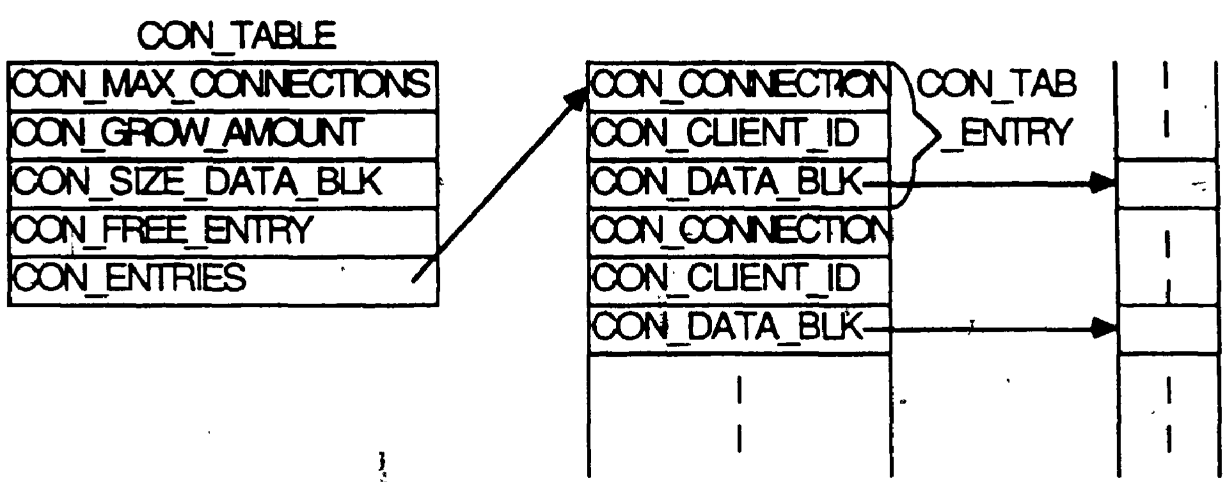


Figure 9.5 Connection Table Manipulated by a Server

init_list. If so it sends an initialization request to _Server_create() again, gets the next record. Then it calls _Report_for_service() to register the secondary name with _Directory.

Upon receiving such a name, _Directory puts it on the head of the secondary_list and replies a registration OK to the server. The server checks the init_list again. If there are more records, it will get one and report it in the same way above, until the init_list empty.

Finally the server exits the initialization loop and calls _Report_for_service() again but this time the passed-in argument msg_type is REPORT_FOR_SERVICE instead of REPORT_SECONDARY_NAME.

Having received the server's primary name, _Directory moves all other names on the secondary_list one by one starting from the list head to the server_list in reversed order. Then _Directory adds the server's primary name to the head of the server_list. This is the original sequence of the records on the init_list. _Directory replies an OK message to the server as usual. The following would be the same as that we discussed in the second case above.

System designer warns that a deadlock can occur if the protocol of "any secondary names must be reported prior to the REPORT_FOR_SERVICE request, and only one REPORT_FOR_SERVICE may be done during a server task's lifetime" is violated.

The first violation causes trouble in the following way. Assume that the REPORT_FOR_SERVICE is done prior to one of REPORT_SECONDARY_NAMEs. The server calls _Report_for_service() and gets an OK message from _Directory. Then it sets up a REPORT_COMPLETED message, sends it to its creator waiting in _Server_create(). The creator misunderstands that initialization has been done and exits _Server_create(). On the other hand, having got the primary name, the server checks the init_list. Since more secondary names exist, it can not exit its initialization loop. So it calls _Send() to send another initialization request to its creator. Unfortunately, its father has left _Server_create(), and won't be able to reply such a request. The server is deadlocked in _Send() primitive within server's initialization loop.

As to the second violation, a server does REPORT_FOR_SERVICE more than once in its lifetime. The first one releases its father from _Server_create(). When doing the second, the server calls _Send() from _Report_for_service() to send a REPORT_COMPLETED message to its father presumptively blocked in _Server_create(). In fact its father was released earlier already by this careless son. The son—server receives the punishment, it deadlocks in _Report_for_service().

Note that these two deadlocks are different in nature from the ones, like the sender ring in message passing. They are caused by violation of the protocol. All these can be verified later in the Petri nets models.

9.2.2 Monitoring Connections

In the server initialization stage, the routine _Alloc_connection_table() may be called to allocate a connection table. Such a table is drawn in Figure 9.5. The number of table entries and size of CON_DATA_BLK are specified in the argument list of _Alloc_connection_table. The initialized CON_CONNECTION fields and their indices are as following :

CON_CONNECTION: 1 2 3 ... CON_MAX_CONNECTIONS;

index: 0 1 2 ... CON_MAX_CONNECTIONS - 1.

_Get_connection() may be called by a server from its case OPEN_REQUEST loop, such as _Tty_server. This routine allocates an entry from the head of the free entry list, referring to Figure 9.6, where the relation

index = CON_CONNECTION - 1

is no longer held in general. CON_CONNECTION field serves as a linking area. CON_CLIENT_ID field is filled with the client id passed in. If the free entry list is empty, this routine will automatically call _Grow_connection_table() to grows a connection table by the CON_GROW_AMOUNT specified in the table.

_Lookup_connection() may be called by a server from its case CLOSE_REQUEST loop for example by _Tty_server. This routine verifies the client and the connection, returns the server's connection data block which is a pointer to char and holds the server's data for connection.

A connection entry no longer needed can be freed by a call to _Free_connection() by a server from possibly its case CLOSE_REQUEST loop. The CON_DATA_BLK is freed and the entry is reinitialized and added to the head of the free entry list (see Figure 9.6).

9.2.3 Open a Connection

To do I/O work, a client first needs to open a connection with the server who is the administrator of the I/O device. Two typical calling diagrams are drawn in Figure 9.7 and Figure 9.8.

A client calls _Open() with the server's name and the mode to use the

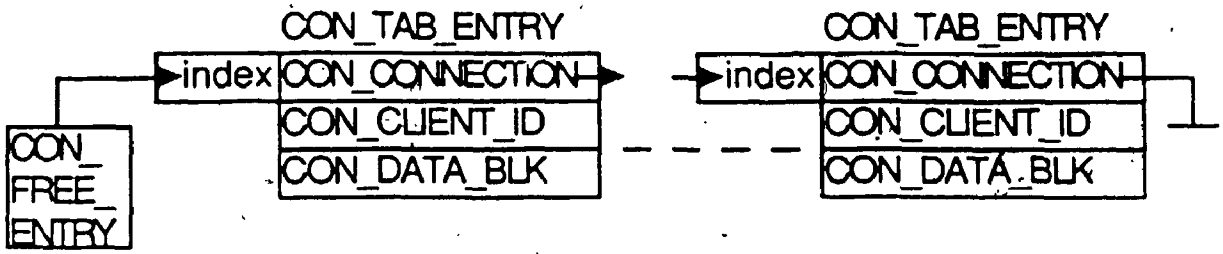


Figure 9.6 The List of Free CON_TAB_ENTRYs

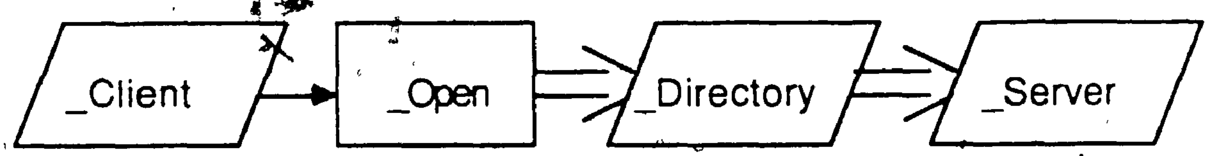


Figure 9.7 Open a Connection

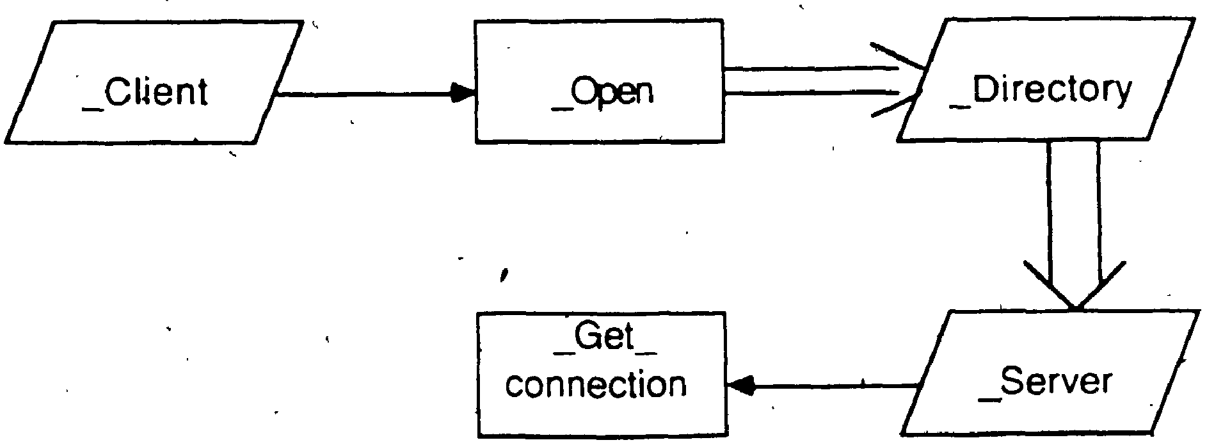


Figure 9.8 Open a Connection

connection parameters specified. Then an open request is prepared and sent to _Directory task. _Directory checks the name submitted through the server_list. If no such a name is found, it will refuse the client's open request by replying a NO_SERVER_FOUND message to the requestor in _Open(). Otherwise it will pass the open request by calling _Send() to the server.

Having received this request, the server usually agrees with it, readily replies an OK approval to the requestor, without any checking or resorting to any other routines like the case in _FD_Format_server. _Aio is such a simple server. The corresponding calling graph is Figure 9.7.

Figure 9.8 is the case for _Tty_server. In server's case OPEN_REQUEST loop, the server calls _Get_connection() where it writes down the client's id and the connection number for this opened stream. The rest would be roughly the same as explained above.

9.2.4 Using Connections (Stream I/O)

A group of functions are provided. The explanations and the examples can be found in the Harmony user manual [3]. They were summarized in the previous section of this chapter as well. Moreover, because they do not involve complicated data structures and synchronizations with other functions, so we skip them here.

9.2.5 Close a Connection

To save the memory space, after I/O service done, a client should close the connection. Two typical calling graphs are shown in Figure 9.9 and Figure 9.10.

A simple case depicted in Figure 9.9 is for _Aio_server. Upon receiving a

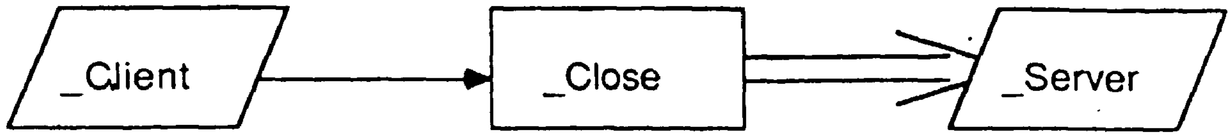


Figure 9.9 Close a Connection

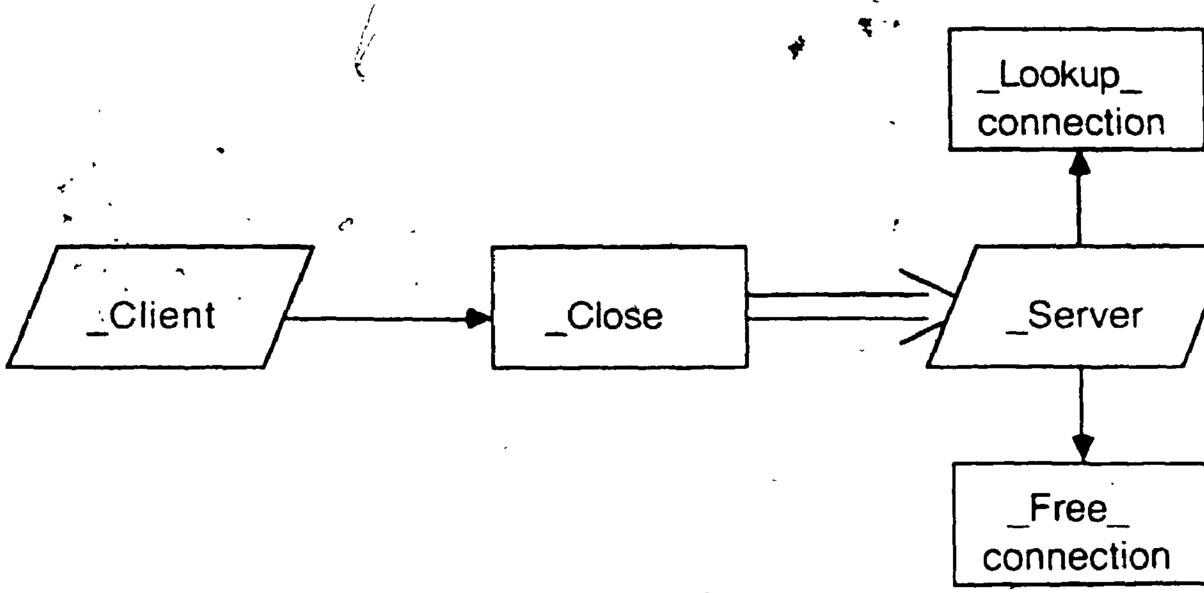


Figure 9.10 Close a Connection

close request, the _Aio_server just approves it, replies an OK to the client in _Close(), as no record for this connection to be closed has been kept.

A more complicated case, such as for _Tty_server, is depicted in Figure 9.10. Because _Tty_server keeps records of opened connections, so when it receives a close request from a client task, it calls _Lookup_connection() to verify such a request, then frees the connection entry in its connection table by a call to _Free_connection(). The rest is quite straightforward.

9.2.6 General Dependency Description

Such a description is well expressed in Figure 9.11. It shows the relationship among the entities involved and described earlier. Not all routines used, but the ones of interest, appear in this diagram for simplicity.

9.3 Detailed Petri Nets Models

The modeling assumptions and notations are the same with those used in previous chapters. We present a concise explanation.

9.3.1 Server Creation, Initialization and Registration

The study can be broken down to subsections according to the number of initialization records the server required.

9.3.1.1 One Initialization Record

To create a server, the user task first prepares a task template for that server and a list of initialization records which are replied to the server one at a time. As a simple case, there is only one record on the initialization list for _Aio_server, as shown in Figure 9.12 and Table 9.1.

A user begins with calling _Server_create() from P_1. The server creation

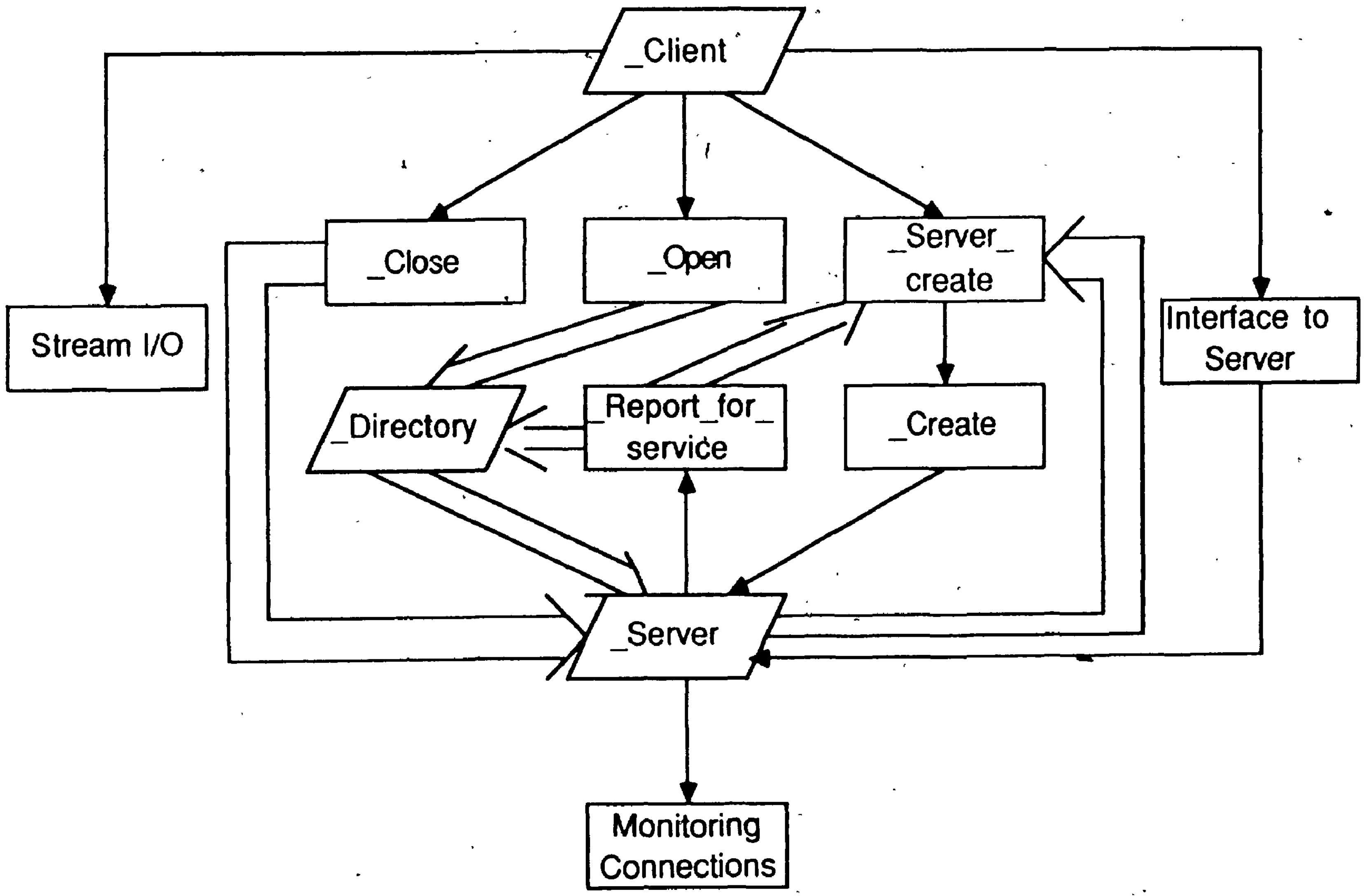


Figure 9.11 Dependency among Tasks and Functions

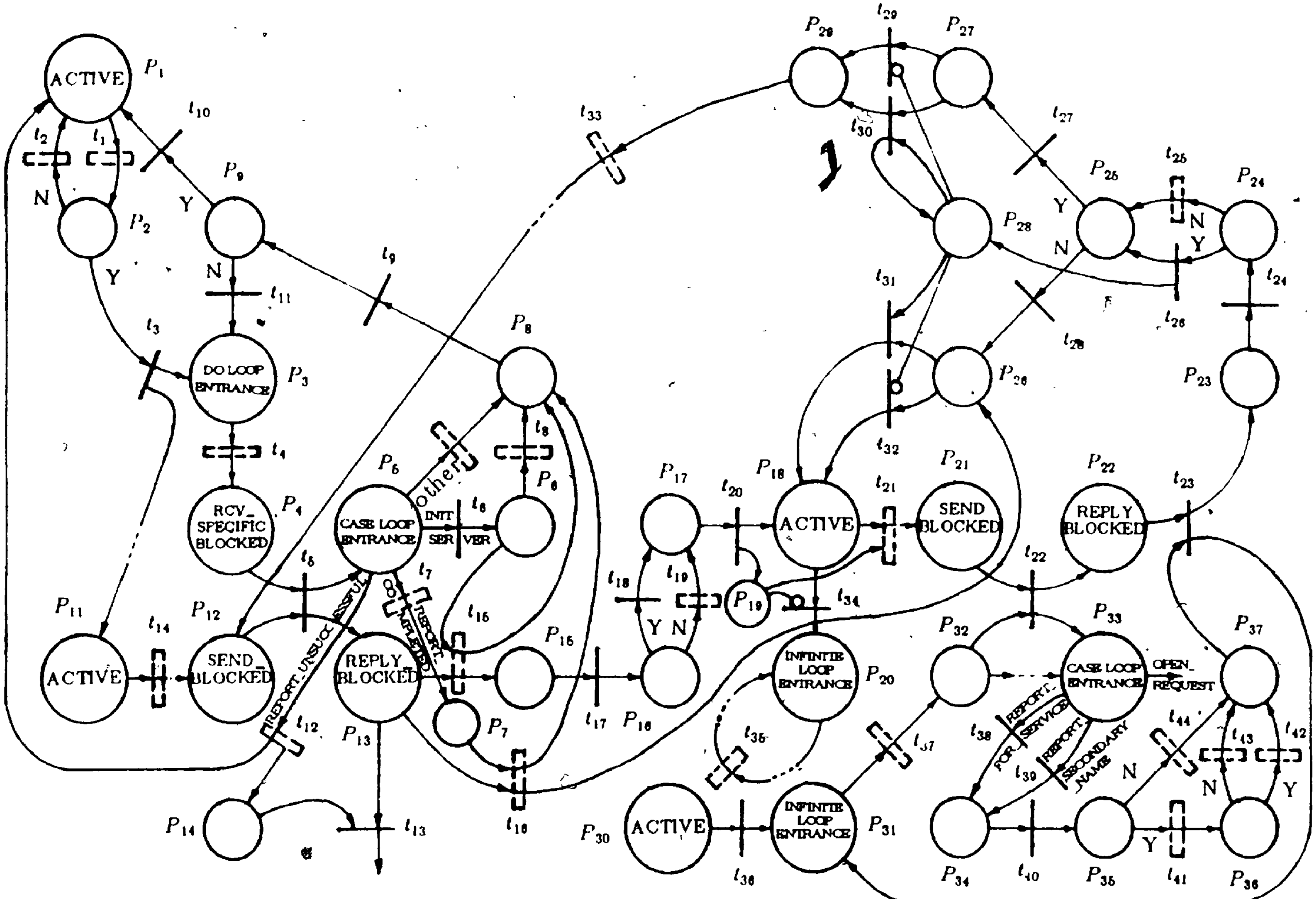


Figure 9.12 Server Initialization and Registration with One Initialization Record

Table 9.1 Server Initialization and Registration with One Initialization Record (To be continued)

Trans.	Meaning
t_1	calls _Create() to create server, is creation successful?
t_2	sets task error code, returns 0
t_3	gets its id and pointers to rqst msg and to a list of init records
t_4	_Receive() from new server
t_5	passes msg to father
t_6	is init_list empty?
t_7	sets up reply msg, _Reply() to server
t_8	displays abortion msg to user
t_9	is rqst msg type REPORT_COMPLETED?
t_{10}	returns server's id
t_{11}	transit transition
t_{12}	sets task error code, kills server, returns 0
t_{13}	removes server out of system
t_{14}	gets dispatched, sets up init rqst msg, _Send() to its father
t_{15}	_Reply() an init record to server, points to next record on init_list
t_{16}	releases server
t_{17}	is a correct init record replied?
t_{18}	transit transition
t_{19}	displays abortion msg to user
t_{20}	takes down replied msg, does REPORT_FOR_SERVICE
t_{21}	sets up rqst msg and reply msg pointer, _Send() to local _Directory
t_{22}	passes msg to _Directory task
t_{23}	releases requestor, _Directory reenters infinite loop
t_{24}	is replied result OK?
t_{25}	sets task error code, is msg_type REPORT_FOR_SERVICE?
t_{26}	is msg_type REPORT_FOR_SERVICE?
t_{27}	gets rqst msg size
t_{28}	transit transition
t_{29}	sets msg type to REPORT_UNSUCCESSFUL
t_{30}	sets msg type to REPORT_COMPLETED
t_{31}	returns 1
t_{32}	returns 0
t_{33}	gets reply msg size, _Send() to father—caller of _Server_create()
t_{34}	server enters infinite service loop
t_{35}	serves clients
t_{36}	initialization
t_{37}	gets rqst msg size
t_{38}	list_ptr points to server_list, moves any other names for this server from secondary_list to server_list
t_{39}	list_ptr points to secondary_list
t_{40}	is reported name valid?
t_{41}	allocates memory for a SERVER_ENTRY, is allocation successful?
t_{42}	writes reported msg to SERVER_ENTRY, adds it to entry list pointed to by list_ptr, set up reply msg, _Reply() to requestor
t_{43}	sets reply msg, _Reply() to requestor
t_{44}	sets up reply msg, _Reply() to requestor
t_{45}	displays abortion msg to user

Table 9.1 Server Initialization and Registration
with One Initialization Record
(Continued from last page)

Place	Meaning
P_1	holds caller of _Server_create()
P_2	fork place for creating the server
P_3	do loop entrance in _Server_create()
P_4	RCV_SPECIFIC_BLOCKED state for the creator
P_5	case loop entrance in _Server_create()
P_6	fork place for testing the init_list
P_7	creator is going to release the server
P_8	join place for the creator
P_9	fork place for testing of completion of server creation
P_{10}	not used
P_{11}	holds _Aio_server
P_{12}	SEND_BLOCKED state for _Aio_server
P_{13}	REPLY_BLOCKED state for _Aio_server
P_{14}	for a control token to remove the server out of system
P_{15}	creator has replied an init record to _Aio_server
P_{16}	fork place for testing correctness of the replied init record
P_{17}	join place for the creator after testing
P_{18}	holds server, caller of _Report_for_service()
P_{19}	for a control token to direct the server
P_{20}	infinite service providing loop entrance for the server
P_{21}	SEND_BLOCKED state for the server
P_{22}	REPLY_BLOCKED state for the server
P_{23}	server has been replied to by _Directory
P_{24}	fork place for testing replied result
P_{25}	transit place for testing replied message type
P_{26}	server is ready to return
P_{27}	fork place for setting message type
P_{28}	holds the OK token
P_{29}	join place for setting message type
P_{30}	holds _Directory task
P_{31}	infinite loop entrance in _Directory()
P_{32}	going to receive a message from the server
P_{33}	case loop entrance in _Directory()
P_{34}	the list_ptr has pointed to a proper server name list
P_{35}	fork place for checking validity of reported name
P_{36}	fork place for checking result of allocating memory
P_{37}	ready to unblock the server

actually is done by a call to _Create() represented by t_1. If the server creation fails, t_2 fires and _Server_create() returns with 0.

If the server creation succeeds, t_3 fires that sets up a pointer to the initialization list. In the meantime, the user enters the do loop and may block in receiving an initialization request from the newly created server, while the server is added to the ready queue and gets dispatched sometimes later.

From P_{11}, _Aio_server sets up the initialization request message and calls _Send() to send the message to its father—the caller of _Server_create() by firing t_{14}. Having received the initialization request, the user enters one of case loops through t_5. If the message type received is not expected, t_{45} fires, an abortion message indicating a fatal error is displayed to the user. The father reentens the do loop gradually and blocks at P_4 RCV_SPECIFIC_BLOCKED which leads to a deadlock because the sender now blocks at P_{13} REPLY_BLOCKED.

Back to P_5, ideally the father should enter INITIALIZE_SERVER case loop, fire t_6 and check whether there are any initialization records left on the init_list. If something went wrong, for instance, the user had not prepared the appropriate number of initialization records, t_8 fires that displays an abortion message to the user. Deadlock occurs later. If the initialization records were well prepared, t_{15} fires, one initialization record is replied to the server. The father returns to P_8, checks whether the type of request message from the server is REPORT_COMPLETED. If not, it will reenter the do loop P_3 to receive the next request from the server.

After firing t_{15}, the server proceeds until fires t_{20} which is the boundary of server initialization and registration with _Directory task.

By firing t_{20}, the server enters P_{18} and prepares to do REPORT_FOR_SERVICE if there is no secondary name for this server on the init_list. Meanwhile a control token is put into P_{19} which enables t_{21} and disables t_{34} so that the server is guaranteed to report for service first.

The server's name and msg_type reported thereafter are packed in the request message and the latter is sent to _Directory task (t_{21}, t_{22}). Next to P_{33}, t_{38} fires, because there is only one name for _Aio_server, the list_ptr points to the server_list. Then memory space is allocated for SERVER_ENTRY at t_{41}. The message from the received initialization record is written to SERVER_ENTRY, the latter is added onto the server_list. Then _Directory replies to the requestor, _Aio_server, all by t_{42}. t_{23} fires. _Directory task releases the server, reenters the infinite loop from P_{31}.

The released server checks the replied result at t_{24}. If registration succeeded, an OK token is generated at t_{26} and added to P_{28}. Then both t_{25} and t_{26} check if the server wants to do REPORT_FOR_SERVICE.

In the case of _Aio_server, the registered msg_type is REPORT_FOR_SERVICE. After P_{25}, t_{27} fires, _Aio_server comes to P_{27}. Then t_{29} and t_{30} set the request message type to REPORT_UNSUCCESSFUL and REPORT_COMPLETED respectively based on the knowledge P_{28} has.

Afterwards the request message is sent to the blocked creator of the server, that is, the caller of _Server_create(). The server goes into P_{13}. If registration fails, msg_type being REPORT_UNSUCCESSFUL, t_{12} fires which kills the ill-born server. The father returns to P_1 with 0. t_{13} fires next, wipes out the server. If another way around, msg_type being REPORT_COMPLETED, t_{16} fires. The father moves to P_8, checks msg_type at t_9. Then t_{10} fires instead of t_{11}. _Server_create returns to P_1 successfully. As another effect of firing t_{16}, the caller of _Report_for_service() comes to P_{26}. At this time, t_{31} is enabled only, because setting request message type to REPORT_COMPLETED at t_{30} earlier was the consequence of firing t_{26} and t_{27}.

Returned to P_{18} with successful initialization and registration, _Aio_server enters P_{20} through only enabled t_{34}. P_{20} is an infinite loop entrance. The loop structure is server dependent. That is the place where the server actually provide services for clients.

9.3.1.2 Several Initialization Records

For some servers, the initialization records may be more than one. Take _Tty_server (terminal server) for example, there are two records on the init_list. The server's primary name is at the first, follows by the secondary name. We explain the algorithm through Figure 9.13 and Table 9.2.

Similar to last model, the created _Tty_server starts from P_{10}. At t_{12} it calls _I_tty_server() that is an agent task responsible for server initialization and registration with _Directory task. Then t_{13} fires. _I_tty_server enters P_{13} infinite loop, sends an initialization request to its father. At t_{15} the first record containing the primary name on the init_list is replied to _I_tty_server. Then t_{16} fires, which creates tti, tto worker tasks, takes down the primary name. t_{18} checks whether the init_list is empty now. In our case, it is not. t_{19} fires next, ... The second initialization record is replied to _I_tty_server. Then t_{17} fires, which calls _Report_for_service() to do REPORT_SECONDARY_NAME. To have more details, let's get back to Figure 9.12, and find P_{33}.

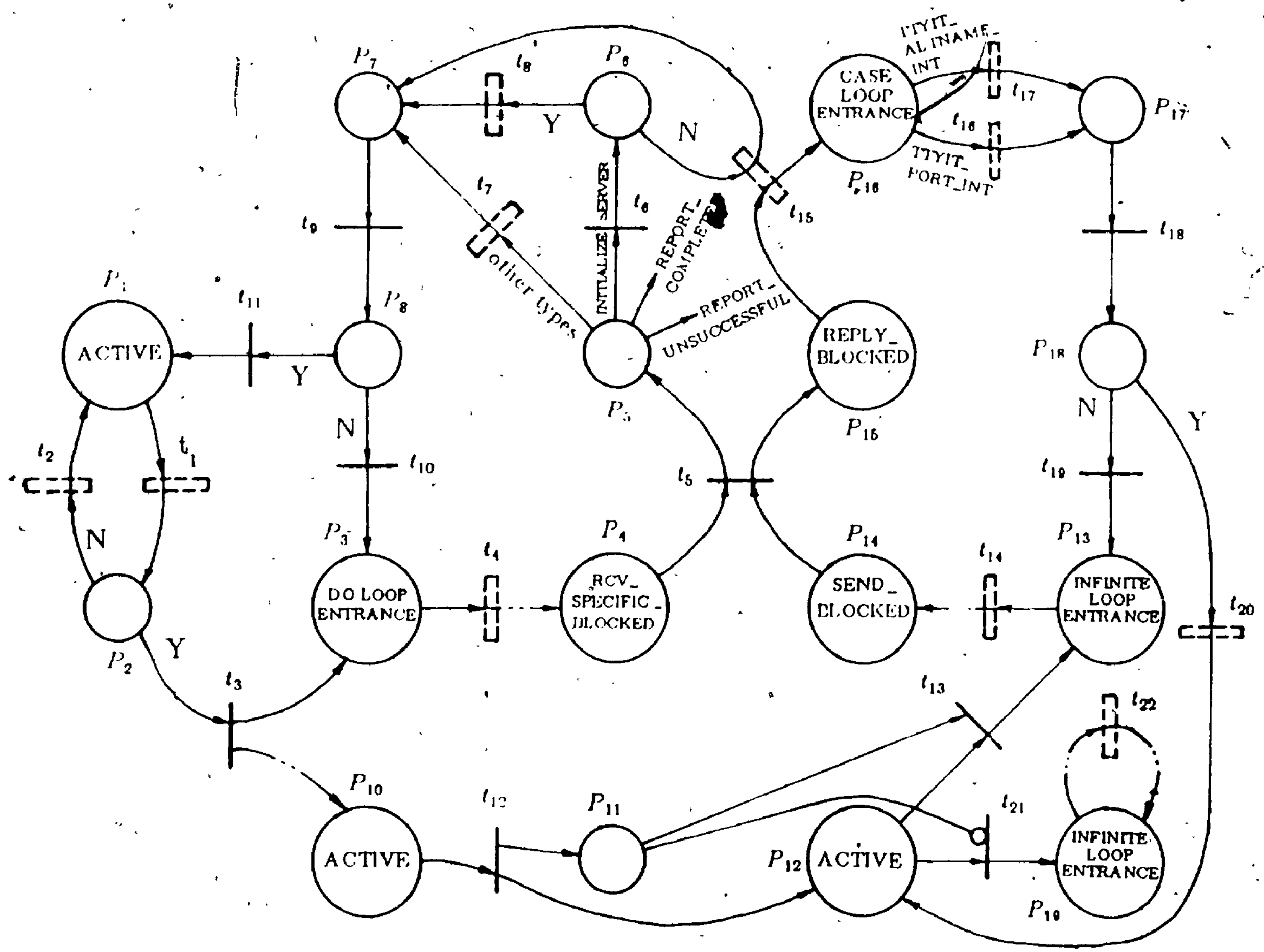


Figure 9.13 Server Initialization and Registration with Several Initialization Records

Table 9.2 Server Initialization and Registration with Several Initialization Records

Trans.	Meaning
t_1	calls _Create() to create server, is creation successful?
t_2	sets task error code, returns 0
t_3	gets its id and pointers to rqst msg and to a list of init records
t_4	_Receive() from new server
t_5	passes msg to father
t_6	is init_list empty?
t_7	displays abortion msg to user
t_8	displays abortion msg to user
t_9	is rqst msg type REPORT_COMPLETED?
t_{10}	returns server's id
t_{11}	transit transition
t_{12}	_Tty_server dispatched, calls _I_tty_server()
t_{13}	transit transition
t_{14}	prepares init rqst msg, sets up reply msg ptr, _Send() to its father
t_{15}	_Reply() an init record to server, points to next record on init_list
t_{16}	creates tti, tto tasks, takes down server's primary name
t_{17}	calls _Report_for_service() to do REPORT_SECONDARY_NAME
t_{18}	has all init records been received?
t_{19}	transit transition
t_{20}	initializes tty_state, calls _Report_for_service() to do REPORT_FOR_SERVICE
t_{21}	server enters infinite service loop
t_{22}	serves clients

Place	Meaning
P_1	holds caller of _Server_create()
P_2	fork place for creating the server
P_3	do loop entrance within _Server_create
P_4	RCV_SPECIFIC_BLOCKED state for the creator
P_5	case loop entrance in _Server_create, here it's INITIALIZE_SERVER
P_6	fork place for testing the init_list
P_7	join place for the creator
P_8	fork place for testing the received message type
P_9	not used
P_{10}	holds _Tty_server
P_{11}	for control token
P_{12}	holds _I_tty_server
P_{13}	infinite loop entrance for initialization
P_{14}	SEND_BLOCKED state for _I_tty_server
P_{15}	REPLY_BLOCKED state for _I_tty_server
P_{16}	case loop entrance in _I_tty_server()
P_{17}	join place for _I_tty_server
P_{18}	fork place for checking completion of initialization
P_{19}	infinite loop entrance in _I_tty_server() for providing services to clients

t_{39} fires, because of the msg_type being REPORT_SECONDARY_NAME. Shortly after, the secondary name is written into a SERVER_ENTRY that is added to the secondary_list at t_{42}. The returned _Report_for_service goes back to P_{18} ACTIVE through t_{23}, t_{24}, t_{26}, t_{28} and t_{31} in Figure 9.12, proceeds to P_{17} in Figure 9.13. If more secondary names need to be replied, t_{19} in Figure 9.13 fires again. The cycle repeats until all initialization records are replied to the server. At that moment, the primary name was taken down, but neither written into a SERVER_ENTRY, nor added to the server_list yet, whereas all secondary names are on the secondary_list.

From P_{18} in Figure 9.13, t_{20} fires next which calls _Report_for_service() to do REPORT_FOR_SERVICE. Back to P_{33} in Figure 9.12, t_{38} fires next, which moves all other names (if any) from the secondary_list to the server_list. Later on, the primary name is written into an allocated SERVER_ENTRY, the entry is added to the head of the server_list at t_{42} in Figure 9.12.

A moment later (still in Figure 9.12), successfully registered server sends a REPORT_COMPLETED message to creator—the caller of _Server_create() to unblock it through t_5, t_7, t_{16}, t_9 and t_{10}, while the server enters the infinite service loop P_{20} and t_{35} gradually. In Figure 9.13, t_{21} fires subsequently. _Tty_server enters the infinite service loop P_{19} and t_{22}.

Now for _Tty_server, both its primary and secondary names are on the server_list. All its SERVER_ENTRYs are removed from the secondary_list. Actually this is true in general.

It is a designing consideration to move all names of a server to the server_list, because later on when a client wants to open a connection with this server, it will check only the server_list.

We can only see the names on the secondary_list temporarily. After all servers' registrations have been done, the secondary_list should be empty. Therefore a doubt is raised whether the secondary_list is a necessity.

In Gentleman's algorithm, servers' names are sent to _Directory task one at a time. Several servers may simultaneously do their registrations with the only one _Directory available in Harmony. Which name can be received by _Directory depends on the result of server's competitions, namely, in random. The snapshots are provided in Figure 9.14. Two init_lists for two servers are prepared and linked by two users as depicted in Figure 9.14(a). At such a moment, all secondary names have been received by _Directory and added to the secondary_list as shown in Figure 9.14(b). The order of names is reversed. The names themselves are interleaved. Upon receiving a primary name from the either init_list, _Directory moves all its secondary names (two here) from the secondary_list to the server_list in a continuous operation thus the names on the server_list are not interleaved. Finally all names from two init_lists are on the server_list in the same order originally linked by the users (Figure 9.14(c)). The secondary_list is empty.

We can drop the secondary_list, and put all names for a server directly onto the server_list in the order linked by the user but not necessarily in consecutive (may be interleaved with other servers' names as depicted in Figure 9.15), thus dramatically simplify the code implementation. It does not add any difficulties to a client when it is searching a server's name through the server_list in _Directory.

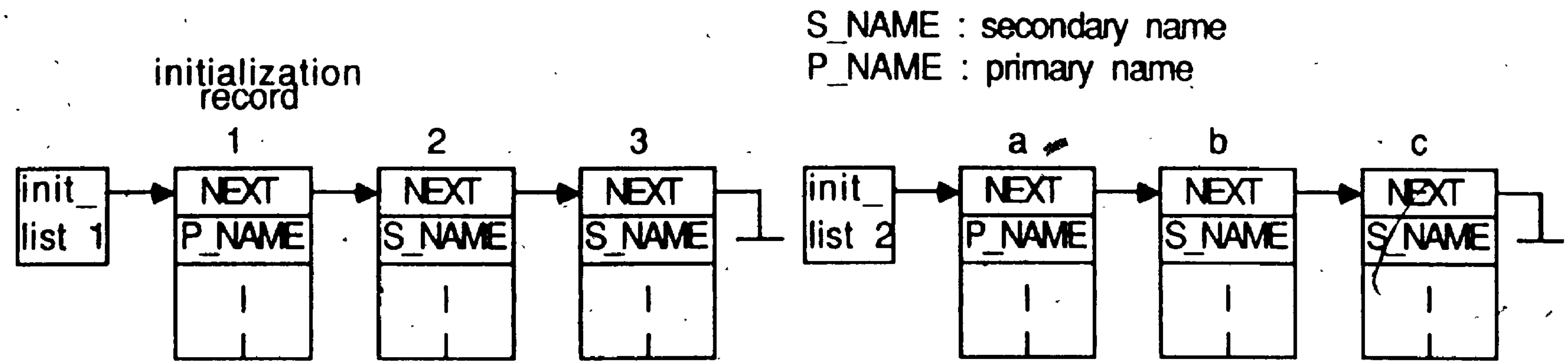


Figure 9.14 (a) Initialization Lists for Servers A and B

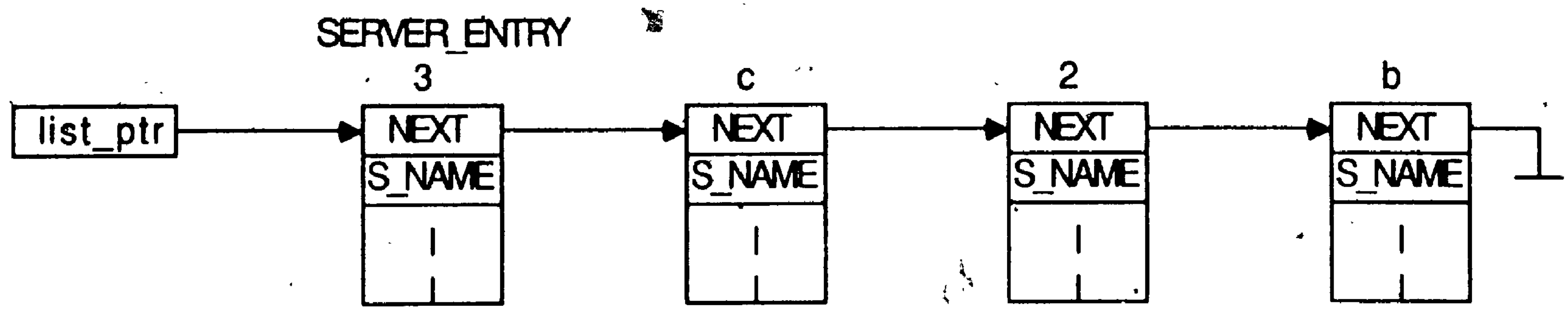


Figure 9.14 (b) The secondary_list Linked by _Directory

P_NAME : primary name
S_NAME : secondary name

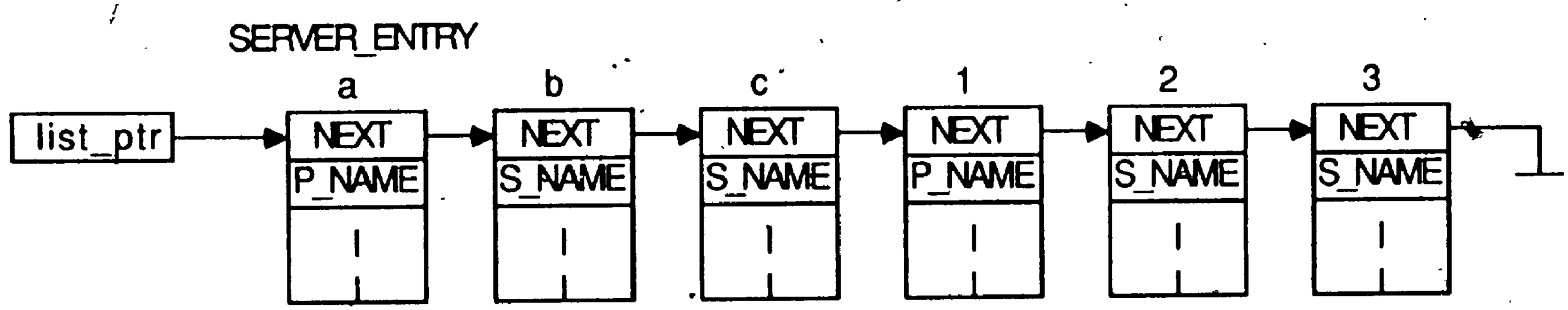


Figure 9.14 (c) The server_list Linked by _Directory

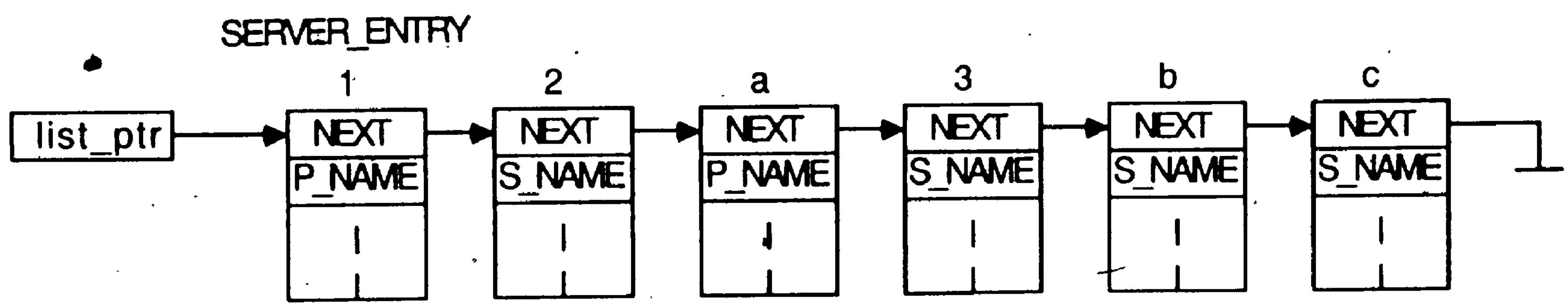


Figure 9.15 The server_list Maintained by _Directory

9.3.2 Open and Close a Connection

Most servers are I/O oriented. Therefore, they must be able to handle the open and close requests.

9.3.2.1 Open a Connection

The PN model is depicted in Figure 9.16 and Table 9.3. A client task calls _Open() from P_1. At t_1 it gets the length of server's name, etc., then sends an open request to _Directory by t_3 and t_4. Having received the open request, _Directory checks the server's name against the server_list at t_{17}.

If no such a name was found, t_{19} fires, so does t_5. _Directory goes back to P_{11} INFINITE_LOOP_ENTRANCE, whereas the client blocked in P_4 is released. Later on, t_7 fires that returns an error message. The open attempt fails.

Back to P_{14}, if the server's name was found, t_{18} fires, which relays the open request. t_{20} passes this message to the server. Here we still use _Aio_server for simplicity.

After having received the open request, _Aio_server replies an OK message to _Directory by t_{21}. In general, a server can check the rest part of the received server's name to decide whether accept or reject a client's open request.

Released _Directory replies the server's decision to the client through t_{22} and t_5. t_5 also frees the memory space for the open request because it is no longer needed. t_6 fires next, which allocates memory space for a ucb. Then t_8 may fire, that initializes the ucb, allocates memory for stream.I.O buffer. t_{10} checks the mode of using the connection specified by the client. If the mode is for the input (read), it sets BUFF_INDEX to BUFFER_SIZE; for the

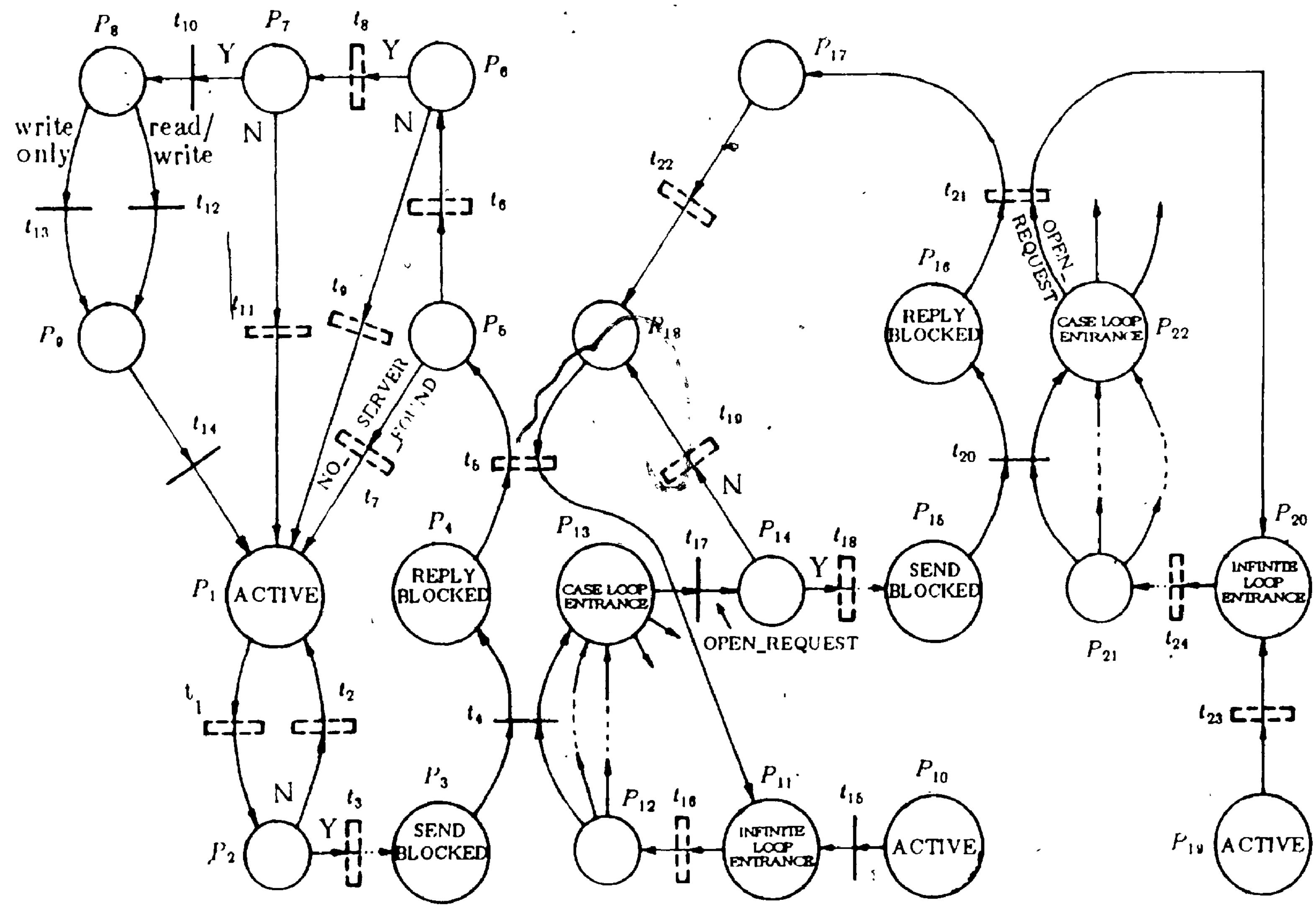


Figure 9.16 Open a Connection

Table 9.3 Open a Connection

Trans.	Meaning
t_1	finds length of server's name, allocates memory for open rqst, checks if allocation is successful
t_2	sets task error code, returns 0
t_3	sets up open rqst msg and reply msg ptr, _Send() rqst to _Directory
t_4	passes msg to _Directory
t_5	unblocked by _Directory, frees memory for open rqst, checks if reply result is OK or not (NO_SERVER_FOUND)
t_6	allocates memory for ucb, is allocation successful?
t_7	sets task error code, returns 0
t_8	init ucb,allocates memory for stream I.O buffer,allocation OK?
t_9	sets task error code, returns 0
t_{10}	check if mode of using connection is read & write (R/W) or write
t_{11}	frees ucb, sets task error code, returns 0
t_{12}	sets BUFF_INDEX to BUFFER_SIZE
t_{13}	sets BUFF_INDEX to 0
t_{14}	sets BUFF_VALID_LENGTH to 0, adds ucb to head of CONN_RESOURCES list, returns address of ucb
t_{15}	initialization
t_{16}	gets rqst msg size, _Receive() any
t_{17}	enters case OPEN_REQUEST loop, checks server name against server_list, server found?
t_{18}	sets up reply msg ptr,sends open rqst prepared by client to server
t_{19}	sets up reply msg, _Reply() it to client
t_{20}	passes msg to server
t_{21}	server enters case OPEN_REQUEST loop, sets up open reply msg, _Reply() it to _Directory
t_{22}	_Reply() to client
t_{23}	initialization
t	gets rqst msg size, _Receive() any

Place	Meaning
P_1	holds client, caller of _Open()
P_2	fork place for checking memory allocation
P_3	SEND_BLOCKED state for client task
P_4	EPLY_BLOCKED state for client task
P_5	fork place for checking replied result
P_6	fork place for checking of allocating a ucb
P_7	fork place for checking of allocating a buffer
P_8	fork place for checking mode of using connection
P_9	join place
P_{10}	holds _Directory task
P_{11}	infinite loop entrance within _Directory()
P_{12}	ready to receive an open request from a client
P_{13}	case loop entrance in _Directory()
P_{14}	fork place for checking if found the server
P_{15}	SEND_BLOCKED state for _Directory
P_{16}	REPLY_BLOCKED state for _Directory
P_{17}	ready to reply to the client
P_{18}	ready to unblock the client
P_{19}	holds _Aio_server
P_{20}	infinite loop entrance in _Aio_server()
P_{21}	ready to receive a message from _Directory()
P_{22}	case loop entrance in _Aio_server()

output, it sets BUFF_INDEX to 0 in order to write data to the output buffer. Finally, a pointer to the opened ucb is returned by t_{14}.

9.3.2.2 Close a Connection

Refer the PN model to Figure 9.17 and Table 9.4. Closing a connection is simple. There is nothing to do with _Directory.

t_1 checks through the caller's connection resource list to find the ucb to be closed. t_3 removes the ucb from the above list and sends a close request to the server. The server usually approves such a request immediately. It replies an OK message to the client by t_5. Finally t_9 frees the buffer and ucb, that is, closes the connection.

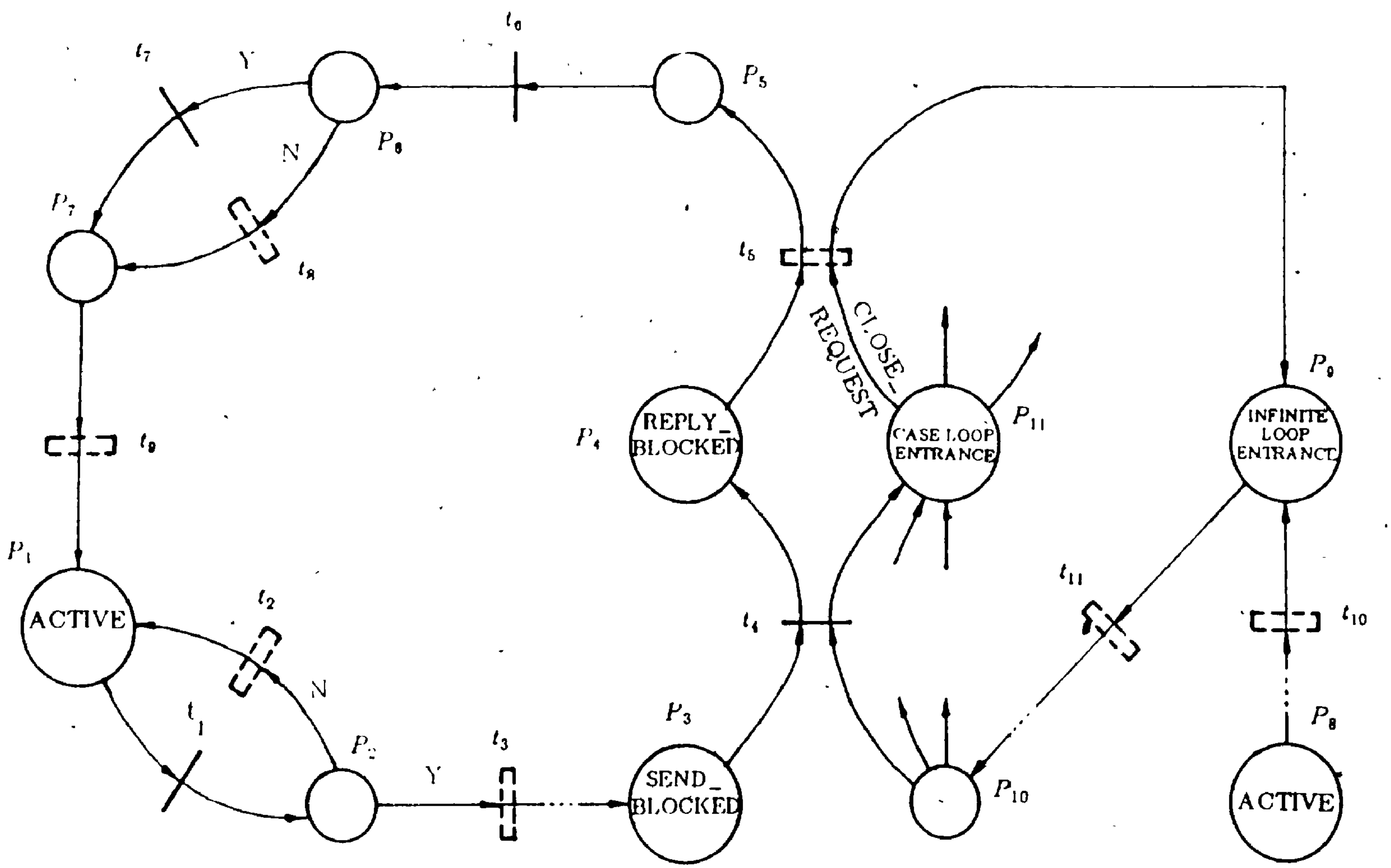


Figure 9.17. Close a Connection

Table 9.4 Close a Connection

Trans.	Meaning
t_1	checks through caller's connection resource list to find the ucb to be closed, has the ucb been found?
t_2	sets task error code
t_3	removes ucb from connection resource list, sets up a close rqst msg, sets up reply msg ptr, _Send() close rqst to ucb server
t_4	passes msg to server
t_5	sets up reply msg, _Reply() to client
t_6	is replied result OK?
t_8	sets task error code
t_9	frees buffer and ucb
t_{10}	initialization
t_{11}	_Receive() any
Place	**Meaning**
P_1	holds client, caller of _Close()
P_2	fork place for checking the ucb to be closed
P_3	SEND_BLOCKED state for a client
P_4	REPLY_BLOCKED state for a client
P_5	the client has been replied to
P_6	fork place for checking the replied message
P_7	join place for the client
P_8	holds _Aio_server
P_9	infinite loop entrance in _Aio_server()
P_{10}	ready to receive a request from the client
P_{11}	ready to enter case CLOSE_REQUEST loop in _Aio_server

PART III

DISCUSSIONS

Chapter 10 Conclusion and Future Work

10.1 Conclusion

In this thesis, the Harmony operating system, excluding various servers, has been modeled by Petri nets. The Harmony source code Release-1 is chosen as the most accurate material which describes Harmony. It is felt that the best modeling approach is one where the model is based on the algorithms and mechanisms described in detailed system documentation, such as the user manual, then have the model confirmed by the source code. In this way, the accuracy is largely guaranteed at both high and low levels. Unfortunately, the system documentation is far from sufficient. In most cases, details have to be obtained directly from the Harmony source code.

In terms of the complexity of the modeling objects, it was easy to build PN models for Chapters 4, 5 and 8 in this thesis. In the system initialization, the PN model using multiple arcs is highlighted by simulating the multiprocessor gates and their values. For the interrupt and error handling, the necessity of PN models may not be impressive. We put them here only for the sake of completeness.

Harmony parts given in Chapters 6, 7 and 9 are so complex that makes PN worthwhile to be used to show its strong modeling power. The message passing is a typical activity which highly involves synchronization and concurrency. The PN model has closely described the algorithm used. The revised PN and discussion on deadlock prevention gave some new ideas.

For the task creation and destruction, two levels, the high and low, PN models were elaborated. The correspondence between them were tabulated. These hierarchical models provide a top-down view of the topic.

As to the server implementation, one of the most lengthy chapters, more attention was given to the description of the algorithms, i.e., the ideas behind the source code by means of data structures, calling graphs, decomposition diagrams. Finally the PN models were refined to precisely describe the mechanisms with the focus on complex ones.

Because the goal is only to model the Harmony by PN, any part of Harmony that involves little or no synchronization and concurrency consequently drew the least or no attention. For this reason, the modeling has covered all major parts of Harmony, but not all of them. For example, the memory management is conceptually important in understanding Harmony. But due to the sequential feature of its code execution, we ruled it out from modeling objects.

As the prerequisite for building PN models, the algorithms have been studied by showing data structures/organizations, calling graphs, decomposition diagrams, and listing functional introductions to program modules. Among them, the depicted data structures and organizations may be most interesting. It might be a pity not to have drawn all important data structures, because documenting Harmony is not the central task of this thesis.

Having the algorithms in our minds, we further expressed them by PN. Here the PN models first serve as a concise and precise description of the algorithms. Sometimes a simple PN model is clearer than a page of explanations. Secondly, it is the summation and abstraction of the source program with emphasis on synchronization and parallelism. Thirdly, the Petri nets themselves provide means of analysis.

10.2 Future Work

PN models have been built up with my great effort. The next question is how to make full use of them.

First, PN models can be easily used for performance analysis. There are variety of analysis techniques available for PN. Among them, the GSPN (generalized stochastic Petri nets) suggested by Marsan [10] is mostly close to the PN used in this thesis. And a software package called GSPNA [9] available here makes solving PN practical.

One obstacle of using GSPNA seemingly comes from the inhibitor arcs extensively used in my models. However, this is not a real obstacle. Actually, the GSPN with inhibitor arcs can be made isomorphic to GSPN without inhibitor arcs, because the inhibitor arcs are reducible.

There are two methods to reduce an inhibitor arc in a PN. First, assign marking dependent probabilities to transitions competing for firing, that is, introduce a random switch (all immediate transitions enabled by a marking together with the associated probability distribution is called a random switch. The associated probability distribution is called a switching distribution). We explain this method by a example shown in Figure 10.1.

In Figure 10.1(a), the firing of immediate transition t_1 or t_2 is controlled by the availability of a token in P_1. We can reduce the inhibitor arc by defining a switching distribution:

$$\begin{cases} Pr(t_1) = 1 - m_1 \\ Pr(t_2) = m_1 \end{cases}$$

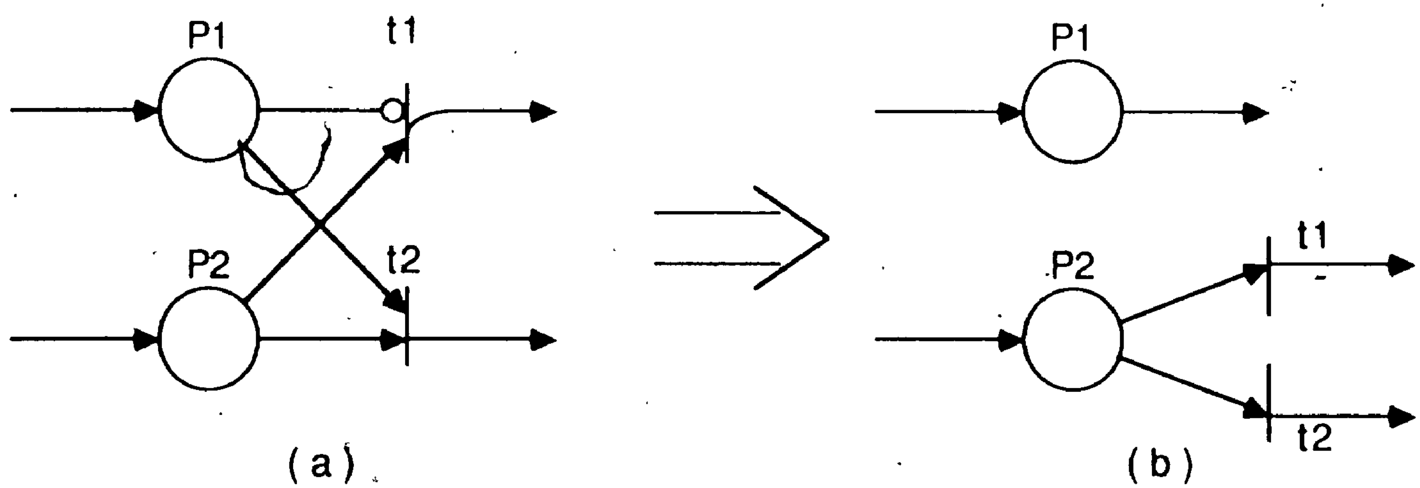


Figure 10.1 Reduction of an Inhibitor Arc by Introducing a Random Switch

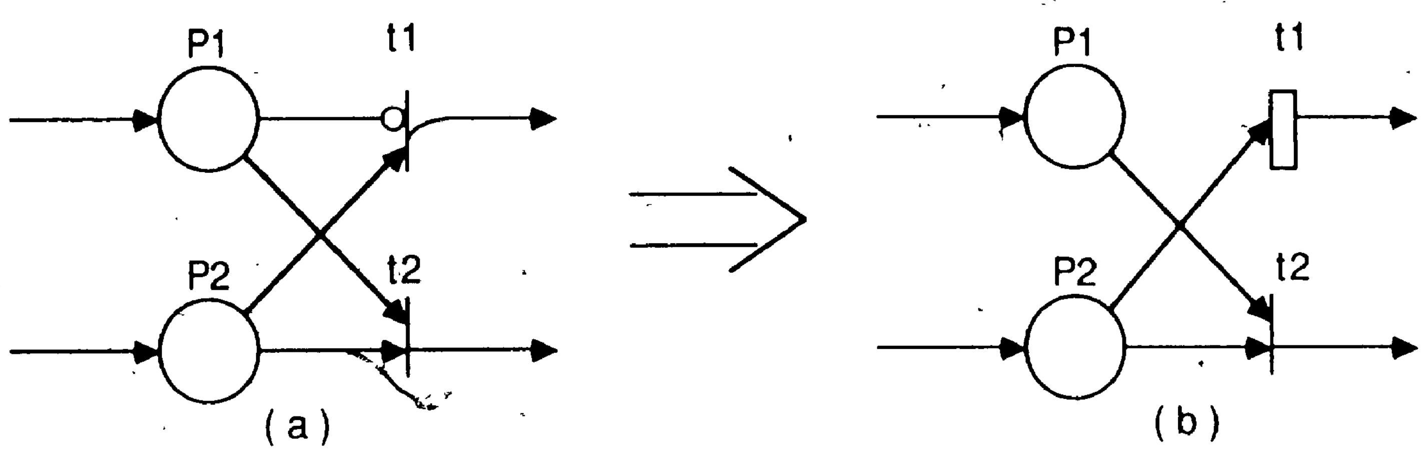


Figure 10.2 Reduction of an Inhibitor Arc by Introducing the Time Transition

where $m_1 = 0, 1$ and is the number of tokens in P_1. In Figure 10.1(b), the mechanism of keeping the number of tokens in P_1 to zero or one is omitted.

The second method makes use of one feature of GSPN—if transitions competing for firing comprise timed transitions and one immediate transition, then only the immediate transition fires. Thus (a) can be reduced to (b) in Figure 10.2. Obviously, method 1 is more flexible than method 2.

The second way of using PN models is the PN analysis. Two major analysis techniques involve the reachability tree and the matrix equations. By using these two techniques, the solution mechanisms can be provided for the problems like safeness, boundedness, conservation, and coverability. Details are available in Peterson's book [16]. Conclusions drawn from such kind of analysis can be used to refine the PN models or improve the Harmony source.

Another application of the PN model is as a tool for the generation of optimal code. To determine the minimal precedence constraints between statements, by using PN model, the individual statements of the program are examined, the artificial sequencing constraints are dropped. Full details are in Shapiro's work [21].

Bibliography

[1] Best, E., "The SOLO Operating System Described by Petri Nets", ASM/8, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, England, August 1976.

[2] Coolahan, J.E. and N. Roussopoulos, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets", *IEEE Trans. on Software Eng.*, Vol. SE-9, No.5, Sept. 1983, pp.603-616.

[3] Gentleman, W.M., "Using the Harmony Operating System", National Research Council Canada, No. 24685, May 1985.

[4] Gentleman, W.M., "Harmony Source Code, Release-0.5", National Research Council of Canada, 1984.

[5] Gentleman, W.M., "Harmony Source Code, Release-1", National Research Council of Canada, 1985.

[6] James, T., K. Rowe, etc., "Experience Porting the Harmony Operating System", ARTT 85-6, Dept. of Systems and Computer Eng., Carleton University.

[7] Lester, B.P., "Analysis of Firing Rates in Petri Nets Using Linear Algebra", *Proc. of the 1985 International Conference on Parallel Processing*, Aug.20-23, 1985, pp.217-224.

[8] Li, Y., "Message Passing in the Harmony Operating System Modeled by Modified Finite State Machine", Technical Report (informal), Dept. of Systems and Computer Eng., Carleton University, April 18, 1986.

[9] Marsan, M.A., G. Balbo, G. Ciardo and G. Conte, "A Software Tool for the Automatic Analysis of Generalized Stochastic Petri Net Models", *Proc. of the International Conference on Modelling Techniques and Tools for Performance Analysis*, Paris, France, May 16-18, 1984, pp.155-170.

[10] Marsan, M.A., G. Conte and G. Balbo, "A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems", *ACM Trans. on Computer Systems*, Vol.2, No.2, May 1984, pp.93-122.

[11] Merlin, P.M., "A Methodology for the Design and Implementation of Communication Protocols", *IEEE Trans. on Communications*, June 1976, pp.614-621.

[12] Molloy, M.K., "On the Integration of Delay and Throughput Measures in Distributed Processing Models", Ph.D. Dissertation, University of California, Los Angeles, 1981.

[13] Molloy, M.K., "Performance Analysis Using Stochastic Petri Nets", *IEEE Trans. on Computers*, Vol. C-31, No.9, Sept. 1982, pp.913-917.

[14] Noe, J.D., "A Petri Net Model of the CDC 6400", *ACM Workshop on System Performance Evaluation*, April 1971, pp.362-378.

[15] Omar, I., K. Rowe and J. James, "An Approach For the Evaluation of Real-Time Operating System", ARTT 85-10, Dept. of Systems and Computer Eng., Carleton University.

[16] Peterson, J.L., "Petri Net Theory and the Modeling of Systems", Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[17] Petri, C.A., "Communication with Automata", Ph.D. Dissertation, Technical Report RADC-TR-65-377, Rome Air Development Center,

Rome, NY, 1966.

[18] Ramamoorthy, C.V. and G.S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets", *IEEE Trans. on Software Eng.*, Vol. SE-6, No.5, Sept. 1980, pp.440-449.

[19] Razouk, R.R. and C.V. Phelps, "Performance Analysis Using Timed Petri Nets", *Proc. of the 4th International Workshop on Protocol Specification, Verification and Testing*, June 1984, pp.126-128.

[20] Sartzetakis, S., P.K. Rowe, etc., "A Real-Time Multiprocessor Performance Monitoring Tool", Report ARTT 85-9, Dept. of Systems and Computer Eng., Carleton University.

[21] Shapiro, R. and H. Saint, "A New Approach to Optimization of Sequencing Decisions", *Annual Review in Automatic Programming*, Vol.6, Part 5, 1970, pp.257-288.

[22] Wan, V.C.F., "A State Machine Description of the Harmony Operating System". Technical Report (informal), Dept. of Systems and Computer Eng., Carleton University, Dec.13, 1985.

[23] Wilson, J., "Harmony's Real-Time Clock Server", ARTT 86-6, Dept. of Systems and Computer Eng., Carleton University.

[24] Zuberek, W.M., "Timed Petri Nets and Preliminary Performance Evaluation", *7th Annual Symposium on Computer Architecture*, 1980, pp.80-96.

Appendix A

Index of Harmony Functions Used

_Abort(s) : §7.1, 74; §8.1, 99

_Add_ready(td) : §6.1, 37

_Addrerr() : §8.1, 104

_Aio_server() : §9.1.6, 110

_Alloc_connection_table(init_num_entries, data_blk_size) : §9.1.4, 109

_Await_interrupt(interrupt_id, rply_msg) : §5.1, 26

_Block() : §3.2, 13; §5.1, 26

_Block_signal_processor(id) : §3.2, 14; §5.2, 32

_Breakpoint(s) : §8.1, 104

_Buserr() : §8.1, 104

_Chkinstr() : §8.1, 104

_Close(ucb) : §7.1, 74; §9.1.2, 108

_Convert_to_td(id) : §6.1, 37

_Copy_msg(from, to) : §6.1, 37

_Create(task_index) : §7.1, 74

_Debug() : §8.1, 99

_Destroy(td) : §7.1, 74

_Directory() : §3.2, 13; §4.1, 19; §9.1.5, 109

_Disable() : §3.2, 14; §5.1, 26

_Em1010() : §8.1, 104

_Em1111() : §8.1, 104

_Enable() : §3.2, 14; §5.1, 26

_FD_Format_server() : §9.2.3, 119

_Flush() : §8.1, 100; §9.1.3, 109

_Free_connection(table, connection) : §9.1.4, 109

_Free_first_block(td) : §7.1, 74

_Free_td(td) : §7.1, 74

_Freevec(block) : §7.1, 78

_Get_connection(table, client, new_connection) : §9.1.4, 109

_Get_td() : §7.1, 78

_Getvec(size) : §7.1, 78

_Gossip() : §3.2, 13; §4.1, 19

_Grow_connection_table(table) : §9.1.4, 109

_I_directory() : §4.1, 19

_Idle_loop() : §3.2, 14

_Idle_task() : §3.2, 13

_Id_Ltm(id) : §7.1, 78

_I_extern() : §4.1, 17

_I_gossip() : §4.1, 19

_I_harmony() : §4.1, 17

_I_idle_task() : §4.1, 19

_Illinstr() : §8.1, 104

_I_ltm() : §4.1, 19

_Infanticide(destroyer) : §7.1, 78

_Invalidate_td(victim) : §7.1, 78

_IP_int() : §3.2, 14; §5.2, 32

_I_ready_queues() : §4.1, 19

_I_stack_and_td(td, stack, stack_start, requestor, root, priority, task_index) : §7.1, 78

_I_store_pool() : §4.1, 19

_I_templates() : §4.1, 19

_I_tty_server() : §9.3.1.2, 128

_I_user_program() : §4.1, 19

_Local_task_manager() : §3.2, 13; §4.1, 19

_Log_gossip(s) : §8.1, 101

_Lookup_connection(table, client, connection) : §9.1.4, 109

_main() : §4.1, 19

_Nointvec() : §8.1, 104

_Open(name, mode, user_id) : §9.1.2, 108

_Privvltn() : §8.1, 104

_Put(byte) : §8.1, 99

_Putstr(s) : §8.1, 99

_Puthex(n) : §8.1, 99

_Receive(rqst_msg, id) : §6.1, 37

_Reply(rply_msg, id) : §6.1, 37

_Report_for_service(name, msg_type) : §9.1.1, 108

_Selectinput(ucb) : §9.1.3, 108

_Selectoutput(ucb) : §9.1.3, 108

_Send(rqst_msg, rply_msg, id) : §6.1, 37

_Server_create(task_index, init_list) : §9.1.1, 108

_Set_task_error_code(error_code) : §8.1, 101

_Setup() : §3.2, 14; §4.1, 17

_Setup0() : §3.2, 14; §4.1, 17

_Signal_processor(id) : §3.2, 14; §5.2, 32

_Sizeof(block) : §7.1, 78

_Spurious() : §8.1, 104

_Stackoverflow() : §8.1, 101

_Suicide() : §7.1, 78

_Task_error_code() : §8.1, 101

_Td_service(id_candidate) : §5.2, 32; §6.1, 41

_Trace() : §8.1, 104

_Trapvinstr() : §8.1, 104

_Try_receive(rqst_msg, id) : §6.1, 41

_Tty_server() : §9.1.6, 110

_Zerodiv() : §8.1, 104

Appendix B

Modified "case UNQ_RECEIVER" in _Td_service(id_candidate)

Gentleman's version :

```
case UNQ_RECEIVER:              /* candidate is sender */
 {
   receiver = _Convert_to_td( candidate→CORRESPONDENT );
   if( receiver )
     {
       /* remove from recv_q */
       p = receiver→TD_NEXT;
       q = receiver→TD_PREV;
       p→TD_PREV = q;
       q→TD_NEXT = p;
     }
   receiver→STATE = ACK_UNQ_RECEIVER;
   _Signal_processor( receiver→ID );
   break;
 }
```

My version :

```
case UNQ_RECEIVER:              /* candidate is sender */
 {
  receiver = _Convert_to_td( candidate→CORRESPONDENT );
  if( receiver )
   {
    /* remove from recv_q */
    p = receiver→TD_NEXT;
    q = receiver→TD_PREV;
    p→TD_PREV = q;
    q→TD_NEXT = p;
    receiver→STATE = ACK_UNQ_RECEIVER;
    _Signal_processor( receiver→ID );
    break;
   }
  candidate→CORRESPONDENT = 0;
  candidate→STATE = READY;
  _Add_ready( candidate );
  break;
 }
```

Appendix C

C Code of Deadlock Prevention in Message Passing

```
*****************************************************************************
   1. Insert deadlock prevention algorithm directly to _Send(), _Receive()
*****************************************************************************
```

Sep 18 15:11 1985 /usr2/harmony/harmony/relea-1/src/kernel/send.c

```
#include ''sys.h''
#include ''kernel.h''
#include ''m68010/kernel.h''

/*
 * $Log:     send.c,v $
 * Revision 1.1  85/08/07  15:40:25  harmony
 * Initial revision
 * Deadlock prevention added.  Yao Li  April 21, 1986
 *
 */

unit_32 _Send( rqst_msg, rply_msg, id )
```

```
char *rqst_msg, *rply_msg;
unit_32 id;
{
extern struct TD *_Active;
extern struct TD *_Convert_to_td();
        struct TD *receiver, *partner;

/* Set up td for receiver */
_Disable();

PRINT(" Send.\n");

/* deadlock prevention */
partner = receiver = _Convert_to_td( id );

while( partner->STATE == SEND_BLOCKED ||
        partner->STATE == RCV_SPECIFIC_BLOCKED ||
        partner->STATE == REPLY_BLOCKED )
  {
   partner = _Convert_to_td( partner->CORRESPONDENT );
   if( partner == _Active )   /* task ring exists */
     {
      if( receiver->CORRESPONDENT != _Active->ID ||
        ( receiver->CORRESPONDENT == _Active->ID &&
        ( receiver->STATE == SEND_BLOCKED ||
          receiver->STATE == REPLY_BLOCKED )))
        {
          _Enable();
```

```
            return( 0 );
          }
        }
      }

    _Active→CORRESPONDENT = id;
    _Active→STATE = SENDING;
    _Active→REQUST_MSG = rqst_msg;
    _Active→REPLY_MSG = rply_msg;

    _Block_signal_processor( id );

    _Enable();
    return( _Active→CORRESPONDENT );
  };

/* Copyright National Research Council of Canada, 1983 */

*******************************************************************************

Sep 18 15:11 1985  /usr2/harmony/harmony/relea-1/src/kernel/receive.c

#include "sys.h"
#include "kernel.h"
#include "m68010/kernel.h"

/*
 * $Log:      receive.c,v $
 * Revision 1.1  85/08/07  15:40:17  harmony
 * Initial revision
```

```
*
* Deadlock prevention added. Yao Li April 21, 1986
*
*/

unit_32 _Receive( rqst_msg, id )
  char *rqst_msg;
  unit_32 id;
{
  extern struct TD *_Active;
  extern struct TD *_Convert_to_td();
        struct TD *sender, *partner, *p;

  if( id )    /* receive specific */
   {
    _Disable();
    PRINT(" Receive Specific.\n");
    /* deadlock prevention */
    partner = sender = _Convert_to_td( id );
    while( partner→STATE == RCV_SPECIFIC_BLOCKED ||
         partner→STATE == SEND_BLOCKED ||
         partner→STATE == REPLY_BLOCKED )
     {
      partner = _Convert_to_td( partner→CORRESPONDENT );
      if( partner == _Active )      /* task ring exists */
       {
```

```
            if( sender→CORRESPONDENT != _Active→ID ||
             ( sender→CORRESPONDENT = = _Active→ID &&
             ( sender→STATE == RCV_SPECIFIC_BLOCKED ||
              sender→STATE == REPLY_BLOCKED )))
             {
              _Enable();
              return( 0 );
             }
         }
    }

   _Active→CORRESPONDENT = id;
   _Active→STATE = Q_RECEIVER;
   _Block_signal_processor( id );

   sender = _Convert_to_td( _Active→CORRESPONDENT );
   if( !sender )
    {
     _Enable();
     return( 0 );
    }
```

****

2. Implement deadlock prevention as a function

****

```
#include "sys.h"
#include "kernel.h"
#include "m68010/kernel.h"

/*
 * $Log:     send.c,v $
 * Revision 1.1  85/08/07  15:40:25  harmony
 * Initial revision
 *
 * Deadlock prevention added.  Yao Li  April 21, 1986
 *
 */

unit_32 _Send( rqst_msg, rply_msg, id )
   char *rqst_msg, *rply_msg;
   unit_32 id;
  {
   char *caller;
   extern struct TD *_Active;
   extern struct TD *_Convert_to_td();

   /* Set up td for receiver */
   _Disable();

   PRINT(" Send.\n");

   /* deadlock prevention */
   *caller = 'S';
```

```
    if( _Deadlock_prevention( caller, id ) = = 0 )
      {
        _Enable();
        return( 0 );
      }

    _Active→CORRESPONDENT = id;
    _Active→STATE = SENDING;
    _Active→REQUEST_MSG = rqst_msg;
    _Active→REPLY_MSG = rply_msg;

    _Block_signal_processor( id );

    _Enable();
    return( _Active→CORRESPONDENT );
  };
```


****

Sep 18 15:11 1985 /usr2/harmony/harmony/relea-1/src/kernel/receive.c

```
#include "sys.h"
#include "kernel.h"
#include "m68010/kernel.h"

/*
 * $Log:      receive.c,v $
 * Revision 1.1  85/08/07  15:40:17  harmony
```

```
* Initial revision
*
* Deadlock prevention added.  Yao Li  April 21, 1986
*
*/

unit_32 _Receive( rqst_msg, id )
   char *rqst_msg;
   unit_32 id;
 {
   char *caller
   extern struct TD *_Active;
   extern struct TD *_Convert_to_td();
        struct TD *sender, *p;

   if( id )    /* receive specific */
    {
     _Disable();

     PRINT(" Receive Specific.\n");

     /* deadlock prevention */
     *caller = 'R';

     if( _Deadlock_prevention( caller, id ) == 0 )
      {
        _Enable();
        return( 0 );
      }
```

```
_Active→CORRESPONDENT = id;
_Active→STATE = Q_RECEIVER;
_Block_signal_processor( id );

sender = _Convert_to_td( _Active→CORRESPONDENT );
if( !sender )
  {
    _Enable();
    return( 0 );
  }
```

.

.

.

****

Sep 18 15:11 1985 /usr2/harmony/harmony/relea-1/src/kernel/deadlockprev.c

```
#include "sys.h"
#include "kernel.h"
#include "m68010/kernel.h"

/*
 * $Log:       deadlockprevention.c,v $
 * Revision 1.1  85/08/07  15:40:25  harmony
 * Initial revision
 *
 * Called from _Send() and _Receive().
```

```
* If a potential task ring exists, returns 0.
* Yao Li  April 21, 1986
*
*/

unit_32 _Deadlock_prevention( caller, id )
  char *caller;
  unit_32 id;
 {
  extern struct TD *_Active;
  extern struct TD *_Convert_to_td();
        struct TD *first_partner, *partner;

  _Disable();

  PRINT(" Deadlock_prevention.\n");

  first_partner = _Convert_to_td( id );
  partner = first_partner;

  while( partner->STATE == SEND_BLOCKED ||
        partner->STATE == REPLY_BLOCKED ||
        partner->STATE == RCV_SPECIFIC_BLOCKED )
   {
    partner = _Convert_to_td( partner->CORRESPONDENT );
    if( partner->ID == _Active->ID )            /* task ring exists */
     {
      if( first_partner->CORRESPONDENT != _Active->ID )    /* big ring */
       {
```

```
            _Enable();
            return( 0 );
          }
        else if( *caller == 'S' &&     /* small ring, caller is a sender */
             ( first_partner→STATE == SEND_BLOCKED ||
               first_partner→STATE == REPLY_BLOCKED ))
          {
            _Enable();
            return( 0 );
          }
        else if( *caller == 'R' &&     /* small ring, caller is a receiver */
             ( first_partner→STATE == RCV_SPECIFIC_BLOCKED ||
               first_partner→STATE == REPLY_BLOCKED ))
          {
            _Enable();
            return( 0 );
          }
      }
    }

  /* no danger */
  _Enable();
  return( id );
};
```

Appendix D

Index of Depicted Data Structures and Organizations

A list of free CON_TAB_ENTRYs (connection table entry) : Figure 9.6, 118

_Comdev (common device) : Figure 5.3, 33

CON_TABLE (connection table) manipulated by a server : Figure 9.5, 114

Data organization for manipulating task templates : Figure 7.4, 80

_Dev_data_table[ ] (device data table) : Figure 5.1, 27

_In_id_q (interrupt id queue) : Figure 5.3, 33

init_list (a list of initialization records for a server) : Figure 9.2, 112

_Int_table[ ] (interrupt table) : Figure 5.1, 27

The linkage of initialization records at different stages : Figure 9.14, 133-134

_Mailbox : Figure 5.3, 33

_MP_gate (multiprocessor gates) : Figure 4.2, 21

Separate priority scheduling system (ready queues for a processor) : Figure 7.1, 73

Three communication queues maintained through a task descriptor (td) : Figure 6.5, 42

Two lists of SERVER_ENTRYs manipulated by _Directory : Figure 9.4, 114

END

09.08.87

FIN