A Path Network Model for Predicting Performance of a Loosely Coupled DRE System in the GeoSurv II UAV Project

by

Zhihong Ke

Supervisor: Professor Trevor W. Pearce

A thesis submitted to

The Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

September 29, 2008

Your file Votre référence
ISBN: 978-0-494-44046-9
Our file Notre référence
ISBN: 978-0-494-44046-9

Abstract

The on-board computer system of the GeoSurv II Unmanned Aerial Vehicle (UAV) is a small-scale loosely coupled distributed real-time and embedded (DRE) system. Ideally, a loosely coupled DRE system is combined with applications, a publish/subscribe middleware, a real-time operating system and time deterministic network hardware. Even though many performance modeling and analysis approaches are available, there is still a need of a simple model to help application developers predict the performance of the GeoSurv II UAV computer system.

This dissertation addresses the need for the development of an analytic modeling approach named Path Network (PN) modeling, which can be used to predict whether a loosely coupled DRE environment is appropriate for the GeoSurv II UAV applications. The visual notation of the PN model is designed, and the temporal parameters of the PN model are characterized. The environment overheads are broken down into several pieces for performance prediction. The algorithm for calculating the performance is described in detail. The prediction results are compared with the measurement results of a UAV experimental system to validate PN modeling. Even though the test-bed is not an ideal loosely coupled DRE environment, the results of the case study still indicate that PN modeling provides a simple and easy way for application developers to predict the performance of the on-board, loosely coupled DRE system of the GeoSurv II UAV.

Acknowledgements

Foremost, I would like to thank my thesis supervisor, Professor Trevor W. Pearce, who shared with me his expertise and research insight. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it. His one-to-one weekly meetings greatly pushed me to work hard and ensured that my dissertation could be finished on time. Without his guidance, encouragement and financial support, this dissertation would not have been possible.

I would like to thank Professor Gregory Franks who kindly provided me a new version tutorial introduction of his LQN model. I would also like to thank Professor Daniel Amyot who provided the graduate course about his UCMs that lights the idea of the visual notation in the thesis.

I would like to thank my classmates Brian Webb and Sean Bassett, who gave me a great help on both academic researching and cultural understanding. Brian helped me correct grammar mistakes in my thesis and always gave me useful advice when I met Linux problems. Sean helped me understand the fundamental knowledge of UAV and his UAV autonomy software.

I would like to express particular thanks to Professor Halim Yanikomeroglu, who provided the critical support to me when I face financial difficulty.

I cannot end without thanking my family, on whose constant encouragement and love I have relied throughout the time at my graduate studies. It is to them that I dedicate this work.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ACK	Acknowledgement
CMU	Carnegie Mellon University
CORBA	Common Object Request Broker Architecture
DAG	Directed Acyclic Graph
DDS	Data Distribution Service
DRE	Distributed Real-time and Embedded
GPS	Global Positioning System
IPC	Inter Process Communication
LQN	Layered Queuing Network
OMG	Object Management Group
PN	Path Network
QN	Queuing Network
RMA	Rate monotonic analysis
RTOS	Real-time Operating System
UAV	Unmanned Aerial Vehicle
UCM	Use Case Map
UML	Unified Modeling Language

Chapter 1 Introduction

Computer systems are continually shaping the world in which we live. A general-purpose computer system is designed to support many different applications, and the flexibility of these systems has led to vibrant markets for desktops, laptops and home computers. The connection of general-purpose computers into distributed networks has become commonplace, as evidenced in the internet and the proliferation of intranets. In contrast, an embedded computer system is designed to support a single application, and the presence of computers in these systems is not usually as obvious to the users. For example, drivers are not usually aware of the presence of computers that may be embedded in the anti-lock braking system of an automobile. Distributed networks are also becoming increasingly common in embedded systems, for example modern aircraft may contain embedded subsystems that are networked to integrate complex functions such as environment sensors, engine operation, control surface actuation, and the pilot interface.

The different objectives of computer systems have led to differences in application development methods and demands on application developers. General-purpose computers are often constructed from general-purpose hardware and include operating systems that separate and hide the hardware from the software applications. This creates a virtual environment for software applications, and they can often be developed without knowledge of the underlying hardware. Furthermore, applications may be interactive but are not usually required to meet time constraints. This simplistic level

of abstraction is not typically practical when developing embedded systems, since applications must interact with application-oriented sensors and/or actuators and the application requirements often include maintaining time constraints among those interactions. The additional demands associated with embedded applications create additional challenges for developers.

The research presented here is motivated by the need to develop an embedded computer system for the GeoSurv II Unmanned Aerial Vehicle (UAV) [34]. The UAV is intended for use in geophysical surveying, and has ambitious operational requirements. The UAV will carry sensitive magnetometers in its wing tips and will fly close to the ground, following the terrain. Autonomous operation is a central goal, as the system will be operated by no more than two persons on the ground. The UAV will automatically fly to a predetermined area, survey the terrain, and then fly back. Obstacles such as towers, buildings and trees must be avoided automatically. These requirements dictate a well-integrated air vehicle and a state-of-the-art avionics system. The GeoSurv II UAV presents a host of engineering challenges, including the development of the application software.

The GeoSurv II UAV is an ongoing project being developed by the Carleton University UAV team. The team is organized as a main group of 4th year engineering students from Department of Mechanical and Aerospace Engineering participating in their final year design project, along with smaller satellite groups of graduate researchers. The goal of the 2008/2009 undergraduate team is to realize the first flight of the first GeoSurv II prototype. While the undergraduate students represent the bulk of the development team, the areas of autonomous operations, obstacle detection, low magnetic

signature actuation system, and low-cost composite structures have been identified as critical to the mission and in need of graduate-level research. The autonomous operations and obstacle detection areas comprise the core of the application software, and researchers from the Department of Systems and Computer Engineering have joined the team to support these areas.

The development of the GeoSurv II UAV faces additional challenges beyond the operational requirements, as the development of the on-board embedded system must be carried out in a very tight schedule. The first flight of the UAV is scheduled for the spring of 2009, and the first vehicle prototype will not be completed until shortly before the first flight. The goal of the first prototype is to demonstrate the ability of the UAV to fly under radio control, and an on-board embedded system will not be required. A prototype that demonstrates initial autonomous capability under radio control is planned for the summer of 2010, with incremental capabilities planned through to the summer of 2012. The final prototype will include an on-board embedded system with software to support autonomous operation and obstacle detection. The development of this system presents a dilemma, since selection of the underlying hardware and software infrastructure to support the application software cannot be made until the application software has been developed to the point that the infrastructure requirements can be estimated accurately. This means that the development of the autonomy and obstacle detection software must carried out without accurate knowledge of the supporting infrastructure, yet must include performance analysis that provides valuable input to the infrastructure selection. Furthermore, the performance analysis should provide confidence that the application will meet its time critical behavior requirements on the final architecture. Ideally,

customized analysis tools could be built to support the development; however, the short timelines are likely to preclude this and the analysis is likely to be done manually.

A further challenge is presented by the relative inexperience of the developers. The undergraduate students participating in the project can contribute to the application software, but have little experience in real-time systems and performance analysis. They cannot be expected to make significant contributions beyond the application software during their limited participation. This means that the development approach must carefully separate the systems-oriented aspects of the on-board embedded system from the application software, and application developers should not be expected to have the skills necessary to contribute towards the systems-oriented aspects. This challenge represents a significant development concern for the UAV team, and is a central issue in the research presented here.

The development strategy for the on-board embedded system is based on the assumption that the final target architecture will be a distributed real-time embedded (DRE) system in which application processes are loosely coupled using a publish/subscribe mechanism. The DRE will consist of a network of embedded computers, and the application software on each computer will be supported by real-time operating system and middleware software that implements the publish/subscribe mechanism. Throughout this document, the computer hardware, operating system and middleware are collectively referred to as the *environment* of the application software. This terminology maintains a convenient distinction between the systems-oriented environment and the application software. The loose coupling provided by the publish/subscribe mechanism is a key factor in assisting application development. This

approach to coupling simplifies the design and analysis of the application, since the publication (sending) and subscription (receiving) of messages is the only interaction method among processes. In other words, the dependencies among processes are caused only by message passing, and no other dependencies such as semaphores, signals, and mailboxes are present. The application software can be easily divided into modules (processes) that can be written by different developers independently. The only interfaces for these modules are the predefined messages that are published and subscribed. The simplicity of this approach is particularly appealing to the GeoSurv II project, where application developers may not have advanced programming skills.

The development strategy also makes extensive use of simulation to address the dilemma discussed earlier. A simulation environment modeled on the architecture will be used to develop and exercise the application software, and environment components will be refined gradually as design decisions are finalized. As a result, the simulation environment will evolve into a platform that is executionally compatible with the target embedded system in the GeoSurv II UAV. In this way, the simulation environment can serve both as a development platform before the on-board DRE system is finalized, as well as a test-bed that is compatible with the on-board DRE system. Furthermore, results from the simulation environment can be used in design decisions about the on-board DRE system.

The development strategy described above has been leveraged by Basset [35] to provide a valuable initial start towards the development of the UAV's autonomy capabilities. He has designed the autonomy architecture for the GeoSurv II UAV based on the assumed architecture, and has developed the design in a simulation environment.

He has chosen the CMU-IPC [8] publish/subscribe middleware to satisfy the loose coupling requirement and for ease of integration, and the use of the middleware has provided a convenient abstraction for developing the application software without concern for its environment. Bassett has demonstrated an initial subset of the required autonomy capability, but his work has not addressed performance analysis.

The goal of the research presented here is to obtain a method for performance modeling and analysis that can meet the needs and challenges of the GeoSurv II UAV project. The method must clearly separate systems-oriented details of the environment from the application software, and must be simple for application developers to apply. Path Network (PN) modeling is proposed as a basis for subsequent refinements.

PN modeling is a performance prediction method designed for a small-scale loosely coupled DRE system based on the CMU-IPC publish/subscribe middleware. The systems-oriented aspects of the environment are separated from the application software, and it is expected that designers with computer systems skills will analyse the environment. Once the environment has been analysed, application developers can use that information in support of their analysis of the application software. PN modeling uses a visual notation to emphasize the aspects of the application and its environment that contribute to application performance. The environment is broken down into overhead parameters that must be analysed by system designers. Applications are modeled in terms of scenarios that involve publishing and subscribing to messages among application processes. Algorithms are provided for calculating scenario response times and CPU utilization in the presence of environment overheads. PN modeling can be used to predict whether a candidate DRE environment is appropriate for the GeoSurv II UAV application

software, and whether a specific deployment of the application software across the environment would meet real-time performance constraints.

The thesis contributes knowledge in three areas. Firstly, it provides a special visual notation for application developers to describe the structural elements and environment overheads in the same graph. Secondly, the environment overhead is broken down into parameters for system designers to measure. Finally, prediction algorithms are provided for calculating the scenario response times and CPU utilization.

This research is the first step in GeoSurv II application performance analysis research, and therefore some simplifications have been made. In particular, the demonstration of the method has been carried out using general-purpose computer systems instead of the DRE systems that would be typical in a UAV implementation. As a result, the performance measurements suffer from inaccuracy due to the non-deterministic time behaviour inherent to general-purpose computer systems. The collection of these measurements and the application of the results in the analysis algorithms are intended only to illustrate the method and are not intended to claim that general purpose computer systems should be used in the UAV. Future research that uses more appropriate DRE systems can be expected to obtain more accurate measurements and therefore provide more realistic input to the development of the GeoSurv II UAV.

The remainder of this document is structured as follows:

Chapter 2 provides a brief overview of the background information about the autonomy research for the GeoSurv II UAV project, DRE systems, publish/subscribe middleware, and performance prediction. Current research in performance prediction and modeling is reviewed in Chapter 3, including related work in visual notations for

performance modeling, as well as research into measuring overheads and delays that influence performance.

Chapter 4 presents the thesis of the research. An analysis of the state of the art research points out limitation in the current modeling approaches used to predict the performance of loosely coupled DRE systems. An ideal solution for performance modeling is then suggested in the context of the GeoSurv II UAV project. PN modeling is then proposed to meet the performance analysis needs of application developers. The contributions of PN modeling towards visual notations, environment overheads and prediction algorithms are then introduced, along with some assumptions and limitations.

Chapter 5 describes PN modeling. The visual notation is described and mapped to the environment overhead model for a loosely coupled DRE system based on CMU-IPC publish/subscribe middleware. Techniques for measuring the environment overheads are presented. Algorithms are also given to calculate two key performance metrics: scenario response time and CPU utilization. Finally, the application of PN modeling is summarized to clarify the roles of system designers and application developers when building and using PN modeling.

Chapter 6 is a case study involving GeoSurv II UAV autonomy. A PN model of the UAV Evasive Manoeuvring Mode is developed and the structural attributes are abstracted from the resulting scenario diagram. The values of environment overheads for a test-bed are measured and calculated. The structural attributes, environment overheads and application processing times are then used as input to the prediction algorithms to generate performance prediction results. Finally, the results of the case study are analyzed in terms of the visual notation, measuring environment overheads, and the prediction

algorithms. To reduce the amount of data presented in this chapter, the measurement data and steps to calculate the temporal parameters in the case study have been moved to Appendix A.

Chapter 7 gives the conclusion of this research and some suggestions for future research.

Chapter 2 Background Information

A brief overview of the background information of this research is provided and some concepts and terms are introduced. Section 2.1 begins by providing an overview of Bassett's research for the autonomy application of the GeoSurv II UAV project. In Section 2.2, the concept of the DRE system is introduced. Section 2.3 describes the publish/subscribe middleware. The concepts of performance prediction are provided in the last section along with the introduction of several current existing performance modeling approaches.

2.1 GeoSurv II UAV Autonomy

Autonomy is commonly defined as the ability to make decisions without human intervention. The responsibility of the GeoSurv II UAV autonomy application is to collect all the information from the sensors, process the data, make the correct decisions, and send commands to the actuators in time. The UAV autonomy should be able to control the UAV while taking off, flying to the destination, surveying the landscape and flying back. It must also be able to control the UAV, so it avoids obstacles and lands safely. A mode diagram is presented here to aid in understanding the behaviour of UAV autonomy.

Figure 1 [35] shows the six modes of the UAV application. These modes are the Launch Mode, the Transit Mode, the Survey Mode, the Evasive Manoeuvring Mode, the Recovery Mode, and the Emergency Recovery Mode. The UAV first enters the Launch Mode, in which the UAV starts up and takes off. When the flight altitude is reached, the UAV enters the Transit Mode. In this mode, the UAV is flying to the survey location. After the UAV arrives the survey location, it enters the Survey Mode and begins to survey the terrain. If the survey has completed normally, the UAV re-enters the Transit Mode and flies to the final destination for landing. The UAV enters the Recovery Mode after it arrives the landing area and ejects its parachute for a safe landing.

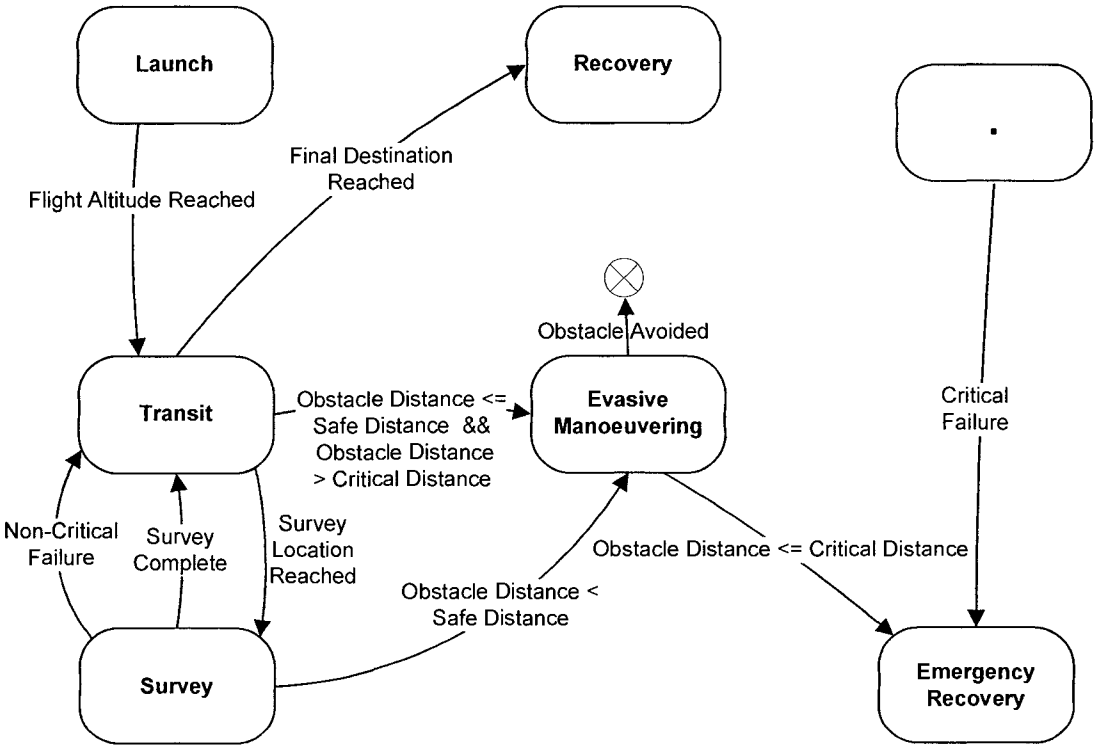

Figure 1: Six Modes of the UAV application [35]

The most critical of the six modes is the Evasive Manoeuvring Mode, or the Obstacle Avoiding Mode. As shown in Figure 2, the UAV flying area usually is divided into three zones: the Safe Zone, the Critical Zone and the Danger Zone. Flying in the Safe Zone means that the distances between the UAV and any obstacle are greater than the critical distance (100m). If the distance between the UAV and an obstacle is less than the critical distance but still greater than the danger distance (50m), the UAV is in the Critical Zone and enter the Evasive Manoeuvring Mode. In the Evasive Manoeuvring Mode, the autonomy continually adjusts the speed, the altitude, and the direction until the UAV leaves the Critical Zone to avoid the obstacle. In some extreme weather conditions, such as strong winds or storms, the UAV may be forced into the Danger Zone. In this situation, the UAV cannot avoid crashing into the obstacle, so the only way to save the UAV is to eject the parachute immediately and land safely. This mode is called the Emergency Recovery Mode. Critical system failures, such as an autopilot failure, also trigger the UAV to enter the Emergency Recovery Mode under which the parachute is ejected to keep damage to a minimum.

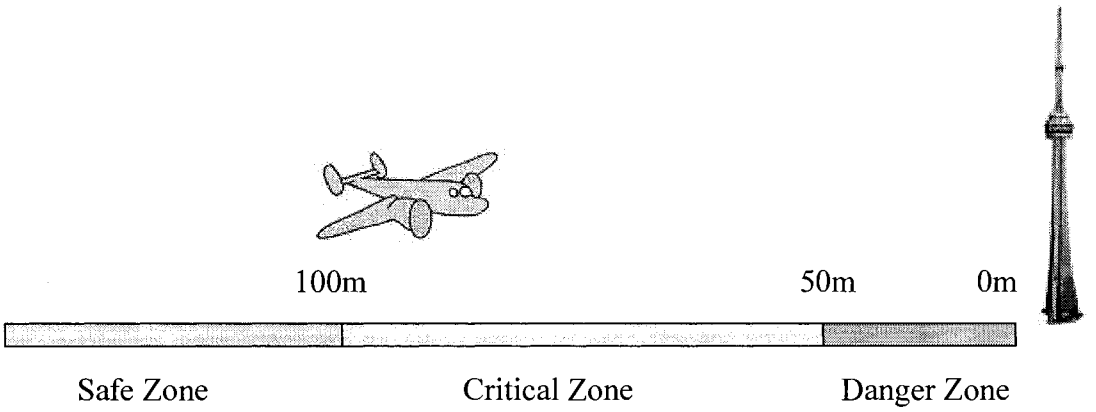

Figure 2: UAV Evasive Manoeuvring

2.2 DRE System

Today, distributed real-time and embedded (DRE) systems are becoming more popular in many areas such as aerospace, military systems, telecommunications, factory automation, robotics, and consumer electronics, because the DRE systems can meet the system requirements of time constraints, resource constraints, modularity constraints, and even constraints of cost, size, and power consumption. The DRE system combines the advantages of the real-time system, the embedded system and the distributed system.

A real-time system is a system in which the correctness of the system behaviour depends not only on the logical result of computations, but also on the time instant at which these results are produced [3]. It is common to divide a real-time system into two types: hard and soft, according to the timing constraints. According to Liu's definition, [2] the timing constraint of a job is hard, and the job is a hard real-time job, if the user requires the validation that the system always meets the time constraint. Validation in this context means a demonstration by a provably correct, efficient procedure or by exhaustive simulation and testing. On the other hand, if no validation is required, or only demonstration that the job meets some statistical constraint suffices, then the timing constraint of the job is soft. The on-board computer system of the UAV is a hard real-time system, because validation of the UAV system is required to make certain the system always meets the time constraint. The response time of the on-board computer system must meet the deadlines to keep the UAV safe and working properly. For example, if the distance between the UAV and an obstacle is less than the critical distance, and the on-board computer system cannot react within the required time to

adjust the speed, altitude, and direction, the UAV has to enter to the danger zone because of a mistake in the design.

Most real-time systems are special-purpose systems designed to perform one or a few dedicated functions. These real-time systems usually also contain the sensors that convert physic signals to electrical signals as a source of the data that will be processed, and the actuators, a subdivision of transducers, that execute commands from the computer systems and control the mechanical equipment. Unlike general-purpose computers, such as personal computers, most computers used in real-time systems do not need keyboards, monitors and printers. They are usually embedded into the system as a part of a complete device. Such an application-specific system is called an embedded system. Since an embedded system is dedicated to specific tasks, system engineers can optimize it, reduce the size, the power consumption and the cost of the product, or increase the reliability and the performance. When an embedded system also has time constraints, it is called an embedded real-time system [3].

Some real-time systems may contain many computers and these computers make up a distributed system. A distributed system is a collection of computers or nodes interconnected by a communication network [1]. The computers do not share memory or a clock. They are not synchronized and they communicate via the message passing. They have a high degree of independency and make autonomous decisions concerning resource access and scheduling. When a system is distributed, embedded, and real-time, it is called a distributed real-time and embedded (DRE) system.

Figure 3 shows the four-layer architecture of an ideal loosely coupled DRE system. These layers are the application layer, the data-centric middleware layer, the real-

time operating system layer, and the deterministic hardware layer. The combination of the last three layers is called the environment, and the combination of the last two layers is called the platform.

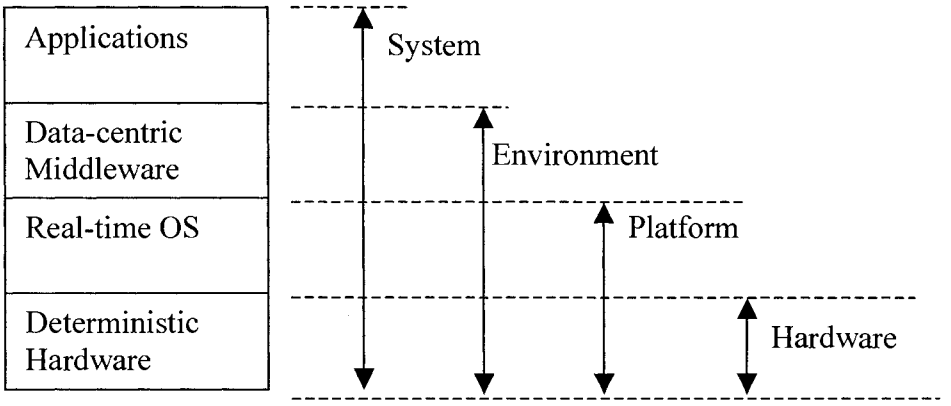

Figure 3: The four-layer architecture of the ideal loosely coupled DRE system

A DRE system usually is a multiprocessing or a multitasking system. The application run on a DRE system can be typically broken down into several processes, so an application is a collection of processes and the process is the basic element of an application. A process is a program in execution and is the combination of several code units that work together to provide some system functions. Such code units are called jobs. All the jobs in a process can share the same memory. The interactions among the processes make the application work. Processes do not share each other's memory. Data exchanging among processes relies on inter-process communications.

In a DRE system, there are two or more nodes. A node represents a computer on the network. All of the nodes connected to the network make up the DRE system. All the process will be deployed onto these nodes. Each node contains several processes. How

the processes are deployed will deeply influence the system performance. When a DRE system contains only a few nodes and a few processes, it can be called small-scale DRE system. Usually a small-scale DRE system can be modeled and analyzed manually without helping with automatic tools. The GeoSurv II UAV on-board computer system is a small-scale DRE system.

2.3 Publish/Subscribe Middleware

Communication middleware software often lies between applications and the operating system, and fills the important role of assisting communication among the processes that run on different platforms of the computer system. There are two main kinds of middleware models: the object-centric or service-centric client/server model and data-centric publish/subscribe model [5]. The middleware for generic enterprise systems, such as CORBA [10] and DCOM [11], often use a tightly coupled client/server communications model. Client/server communications is characterized by a network resource (the server) that other network nodes (clients) access to obtain data or perform functions. Communications begin with a request by a client and end when the server replies [6]. In contrast, most DRE systems are data-centric, as the applications require data distribution, not remote service invocations or function calls. Using a client/server model, such as CORBA, on a data-centric DRE system frequently leads to unnecessarily complex systems and poor system performance [6]. As a result, the middleware for the DRE system often uses a different model, called the publish/subscribe model to perform inter-process communication.

Publish/subscribe communications middleware, such as OMG's Data-distribution Service (DDS) [9], SUN's Java Message Service (JMS) [21] and CMU-IPC are nowadays considered a key technology for information transmission on DRE systems. Each participant in a publish/subscribe communication system can play the role of a publisher (message producer) or a subscriber (message consumer) of information. Publishers produce information in the form of events, which are then consumed by subscribers [20]. The Publish/subscribe communication also provides subscribers with the ability to register their interest in messages generated by publishers. A message is a data structure sent by one process and received by another process. The trace of a certain message is called its message path and a sequence of message paths is called a scenario. A scenario shows a clear message trace from event to effect.

Publish/subscribe middleware provides a loosely coupled form of interaction between the message producer and consumer. There are three kinds of decoupling between a publisher and a subscriber [7]:

- Space decoupling: Both publisher and subscriber do not need to know each other.
- Time decoupling: Both publisher and subscriber do not need to be actively participating the interaction at the same time.
- Synchronization decoupling: A publisher is not blocked by any subscriber after producing message, and any subscriber can obtain the message asynchronously.

CMU-IPC [8] (Inter-Process Communication) is an open source publish/subscribe middleware that is developed by Reid Simmons of Carnegie Mellon University. CMU-IPC is designed to facilitate communication between heterogeneous control processes in

an engineered system. A CMU-IPC-based system consists of an application-independent middleware daemon and several application-specific processes.

The daemon has a key role in the CMU-IPC middleware. The daemon normally performs a store-and-forward function to route messages from publishers to subscribers. The daemon is also a repository for system-wide information (such as message names, process names) and can be located at any node. All the publishers and subscribers in the system should register with the daemon the messages that they want to send or to receive. When a publisher publishes a message, the daemon routes it to its subscribers. After the daemon sends the message to a subscriber, it is self-blocked. When the subscriber gets its turn to run, the subscriber receives the message and sends back an acknowledgement (ACK) to the daemon to indicate success in receiving. The daemon wakes up when it obtains the ACK. Then the daemon self-blocks again, and waits for receiving next message.

If two or more subscribers register the same message, the CMU-IPC middleware daemon sends copies of the message to each subscriber on a first-come-first-served basis according to the sequence they registered. A sending sequence relies on the run-time of the process registration is not a deterministic design and not suitable for a real-time system. An improvement could be made on the IPC middleware by connecting the sending sequence of message copies with the process priority, meaning the higher priority process obtain its message copy first.

The CMU-IPC middleware provides timers that enable user-specified functions to be invoked at a given point in time, or periodically over a given interval. The timers enable a module to perform time-critical actions, or to dispatch events at specific times.

The timers can be used for time dependent operations, but they are not truly interrupt-driven. The timers will not be accurate if a given time interval is less than 20 ms.

The CMU-IPC middleware was used in the early stage of research for the GeoSurv II UAV project, and was also selected as the middleware of the target system of this dissertation. Because CMU-IPC is a non-real-time middleware, a more real-time middleware must be chosen in the final on-board DRE system.

2.4 Performance Prediction

Performance prediction means to estimate the values of performance metrics of an application on a given computer system at the development stage. A metric is a standard unit of measurement, for example: length, area and speed. A performance metric is a unit standard of measurement that is used to assess the performance of a system. There are many kinds of performance metrics for computer systems, such as response time, CPU utilization, memory contention, network bandwidth and throughput [13]. Only the response time and the CPU utilization are of concern for this dissertation.

Response time metrics measure the time interval between the initiating or requesting some action and the arrival of the result. Response time is different from executing time. Executing time is the time that a task would take if it were to execute without interference. The response time is also different from the deadline. The deadline is the length of time during which the task's output would be valid in the context of the specific system. If all response times are less than the deadlines that the requirement demands, the system is usually feasible. The scenario response time is defined as the

response time for a whole scenario. It is a time interval from the moment that the scenario is triggered by a sensor device to the moment that the scenario triggers an actuator device.

The CPU utilization metric is the ratio of the time that a CPU is busy divided by the time that the CPU is available. CPU utilization is a useful metric for evaluating computer performance. Laplante's work [4] provides recommendations for the different ranges of utilizations that are shown in Table 1. The CPU utilization cannot greater than 100%. When the utilization range is 83% to 99%, the system is running in a heavy load and is dangerous. Some processes may miss deadlines in this situation. In contrast, the utilization range from 0% to 25% indicates that such design wastes CPU power and money. In the range from 26% to 68%, the system is in the safe zone.

Utilization (%)	Recommendations
0-25	Waste CPU power and money
26-50	Very Safe
51-68	Safe
69	Theoretical limit
70-82	Questionable
83-99	Dangerous
100	Overload

Table 1: CPU utilization zones and recommendations [4]

The system performance is influenced by system structural attributes and system temporal parameters. The structural attributes represent the relationship among the structure elements, such as processes, messages and nodes. The structural attributes includes sets of processes, messages or nodes that satisfy certain conditions, such as a set

of processes that belongs to a given scenario and a set of processes whose priority is higher than the priority of a given process in a given node. A graphic model helps collect the structural attributes.

The temporal parameters, such as processing time and environment overheads, contribute the response time or CPU utilization. The processing time is the time required for message handlers to complete their functions, such as the processing of messages, when they execute alone and have all the resources they requires. Processing time does not include any overhead. The environment overhead is defined as the total time interval cost in the DRE environment, such as message sending overhead, message receiving overhead, context switch overhead, and middleware daemon service overhead, etc. All temporal parameters should be considered in performance prediction.

Delays should also be considered when predicting the response time of a scenario. The higher priority processes often cause the delay of lower priority processes. In a given node, if there is more than one process ready to run, lower priority processes always wait for the completion of higher priority processes. The daemon can also delay the current running process, if the daemon is set to a higher priority. Both delays influence the response times of scenarios, and should be taken into account to obtain the pessimistic result.

The pessimistic performance prediction is much more useful for hard real-time systems, like the UAV, than the optimistic prediction. This is because the user requires the validation that the system always meets the time constraint. The pessimistic prediction results are larger than the performance measurement results. Therefore, if the

pessimistic prediction results still meet the performance requirement, the developers may gain more confidence on the application software.

A graphic performance model can help application developers better understand the possible influence of different kinds of structure attributes and temporal parameters in the performance prediction. A model is an abstraction or a generalized overview of a real system and helps application developers to better understand the system under development. There are two major types of performance models: the analytic model and the simulation model [13]. Analytic models are composed of a set of formulas that provide the desired performance metrics as a function of the set of input temporal parameters. Simulation models are based on computer programs that emulate the different dynamic aspects of a system as well as their static structure. Analytic models are less expensive to construct and tend to be computationally more efficient to run than simulation models. Simulation models can be made as detailed as needed and can be more accurate than analytic models. There are some system behaviours that analytic models cannot (or very poorly) capture, thus necessitating the need for simulation [13]. Many analytic modeling approaches are used in performance prediction, such as Petri Nets, Queuing Networks (QN), Layered Queuing Networks (LQNs), Unified Modeling Language (UML), and Use Case Maps (UCMs) etc.

Petri Nets [12], as a graphical and formal mathematical modeling language with a well-defined syntax and semantics, provide a uniform environment for modelling, formal analysis, and design of discrete event systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behaviour of systems. Petri Nets consist of places, transitions, and arcs that connect them.

Input arcs connect places with transitions, while output arcs start at a transition and end at a place. Places can contain tokens. Transitions are active components and transitions are only allowed to fire if they are enabled. When the transition fires, it removes tokens from its input places and adds some at all of its output places. To study performance and dependability issues of systems it is necessary to include a timing concept in the model. There are several extensions of Petri Net that do this; however, the most common way is to associate a firing delay with each transition. This delay specifies the time that the transition has to be enabled for, before it can actually fire.

Lazowska et al. [15] have described Queuing Network (QN) modelling, an analytic modelling approach to computer systems, in their book published in 1984. In QN modeling, the computer system is represented as a network of queues. A network of queues is a collection of service centres and customers. To define a software-level QN model, the customer represent user requests and service centres represent software processes. The queue delay at each centre is an estimate of time that the requests are blocked awaiting access to the corresponding process.

Layered Queuing Network (LQN) modeling is an important extension of QN modeling, which was designed by Woodside, and Franks [17]. An LQN model is represented as a directed acyclic graph (DAG) with two types of nodes: tasks (software entities, drawn as parallelograms) and processors (representing all types of hardware devices, drawn as circles). Arcs represent service requests, either synchronous (when the client makes the request, it blocks until it receives a response from the server) or asynchronous. In an LQN model, the resources are ordered into layers, typically with user processes near the top and hardware at the bottom. There are three kinds of interaction

styles between two processes in LQN modeling. The first style is an asynchronous call, in which the sender does not wait and receives no reply. The second style is an synchronous call, in which the sender waits (blocked) for the reply, and the receiver must provide replies. The third communication style is forwarding interaction, which allows for a client request to be processed by a chain of servers instead of a single server. The request is forwarded among the servers until the last server replies to the client. Each server in the chain becomes idle as soon as it has completed its job. In each interaction there is a calling party taking a "client" role and a called party in a "server" role.

The Unified Modeling Language (UML) is a standard language for writing software blueprints [14]. The UML is appropriate for modeling systems ranging from enterprise information systems to distributed web-based applications and even to hard real-time embedded systems. The UML includes thirteen kinds of diagrams that can be separate into two categories: structure diagrams and behaviour diagram. Structure diagrams emphasize what things must be in the system being modeled. Behaviour diagrams emphasize what must happen in the system being modeled. The deployment diagram and the sequence diagram are discussed and used in this dissertation. The deployment diagram shows the configuration of run time processing nodes and the components that live on them. It addresses the static deployment view of the architecture. The sequence diagram shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them. It emphasizes the time ordering of messages.

Amyot et al. [31] utilize the visual notation Use Case Maps (UCMs) for capturing the requirements of reactive and distributed systems. UCMs use behaviour as a concrete,

first-class architectural concept. They describe scenarios in terms of causal relationships between responsibilities. UCMs usually emphasize the most relevant, interesting, and critical functionalities of the system. They can have internal activities as well as external ones. UCMs are abstract (generic), and could include multiple traces. With UCM, scenarios are expressed above the level of messages exchanged between components; hence they are not necessarily bound to a specific underlying structure. UCMs provide a path-centric view of system functionalities and improve the level of reusability of scenarios.

Current research in performance prediction that uses the above modeling approaches is reviewed in the next chapter. This includes related works on visual notations for performance modeling, as well as research into overheads and delays that influence the performance. Work done on the measurement of environment overheads and the performance prediction algorithms is also outlined.

Chapter 3 State of the Art

In the literature, there are lots of well-known and widely used performance prediction and modeling approaches for DRE systems. Some of them analyze the overheads on the DRE system, and others provide visual notations to meet different kinds of needs. In Section 3.1, research in the area of visual notations for performance modeling is described. Section 3.2 outlines the current research in system environment overhead measurement. The works related to the performance prediction algorithms are introduced in the final section.

3.1 Visual Notation

Zanolin et al. [24] describe an approach to support the modeling and validation of publish/subscribe architectures. A UML state-chart diagram is used to describe states of each process, where transitions describe how the process reacts upon the receipt of a message. Transition labels are used to show the precondition and action of the message publishing. Live Sequence Chart (LSC) is an extension of Message Sequence Chart (MSC), which looks like a UML sequence diagram. LSC is used to describe how the publish/subscribe architecture behaves. Messages exchanged between processes are drawn as arrows and are asynchronous by default. Each message is decorated with a label that describes the message itself. The middleware is omitted in the charts. An arrow line between two processes implicitly indicates that the message is first sent to the middleware and then routed to the other entity. In the example of LSC shown in Figure 4,

the process 1 publishes the message 1 to the process 2, and the process 2 is triggered and publishes the message 2 to the process 3. The sequence chart can describe the relationship between the processes and messages and the possible scenario of a message flow. The dependency among messages also can be shown along with the scenario. It can clearly indicate the message publish/subscribe mechanism.

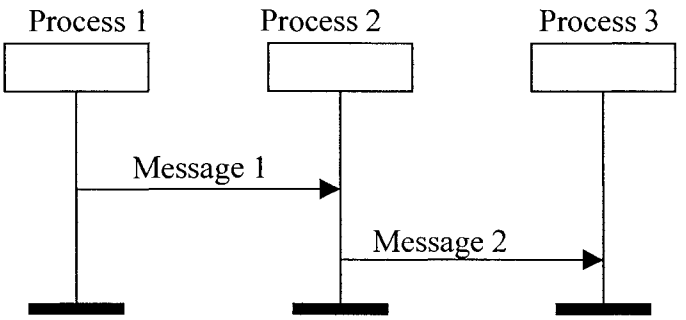

Figure 4: An example of the Live Sequence Charts

Ti-Yen Yen et al. [18], describes methods for the performance analysis of an embedded application executing on multiple processors. A fixed-point iteration algorithm is used to compute tight bounds on the worst-case delay through a task graph executing on multiple processors, including complications caused by data dependencies in the task graph model. A task graph, which is a Directed Acyclic Graph (DAG), is used to represents the structure of a set of processes in their work. Figure 5 shows an example of a task graph. A directed edge from the process P_i to P_j represents that the output of P_i is the input of P_j. A vertex in the task graph is a process. A vertex could have one or more inputs. A process is not initiated until all its inputs have arrived; it issues its outputs when

it terminates. The task graph describes the dependency among the processes, it is a simple graph and can be drawn easily.

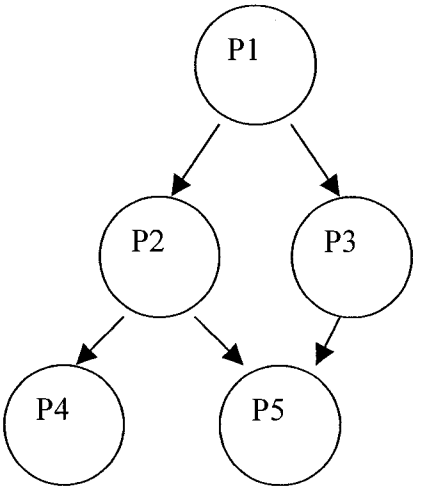

Figure 5: An example of task graph

Rajkumar et al. [25] provide a real-time publisher/subscriber communication (RT-PSC) model for distributed real-time systems. The model has been used successfully in building the software architecture for an upgradeable real-time system. The IPC model in Figure 6 shows the relationship of all structure elements at different levels. These elements include nodes, IPC clients (processes), IPC daemons (middleware), threads, and the real-time operating system. A message is represented by a double arrow line. Application-level clients (processes) use the communication model by making calls that are implemented by a library interface hiding the complexity of the underlying implementation. IPC daemons on various nodes communicate with one another. This RT-PSC model clearly describes the deployment of the processes in the nodes (IPC clients),

and it also shows the roles of daemons. The threads used for publish/subscribe are also shown inside the processes and daemons.

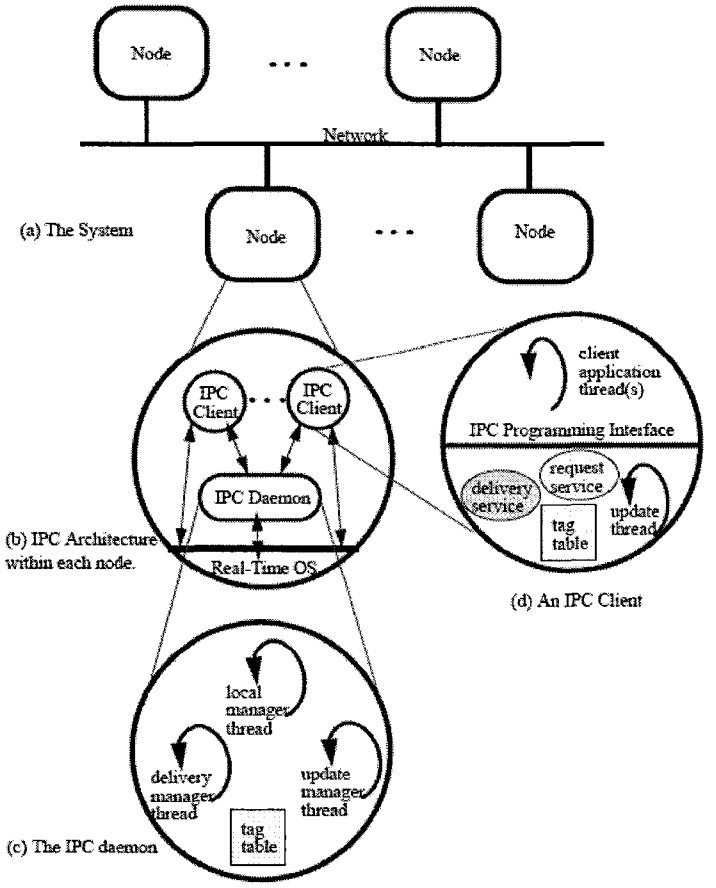

Figure 6: The Architecture of RT-PSC Model [25]

Verdickt et al. [27] use Layered Queuing Networks (LQN) to model the performance of CORBA middleware. The LQN modeling is supported by a toolset that contains analytical solvers and a simulator to extract performance estimates from an LQN model. LQN is an extension to the widely used QN model. The most important difference between LQN and QN is the fact that a server serving a client request can become a client of another server, thus modeling nested services and synchronous calls. LQN supports

multi-entry, multi-activity, and multi-thread processes, and allows different service times for same process. All entries have their own execution times (specified per phase) and can make calls to other entries. The number of calls can be fixed or can vary with a given mean and standard deviation. The network also can be divided into multiple entries. This models the fact that the messages and the responses have different sizes and thus take a different amount of time to travel through the network, and make the prediction result more precise. Figure 7 presents a small example of an LQN model of a web server. Users request web pages from a user browser to a web server. The requests may be queued before the web server processes them. The server requests the disk fetch the page the user wanted, and there is delay caused by the network.

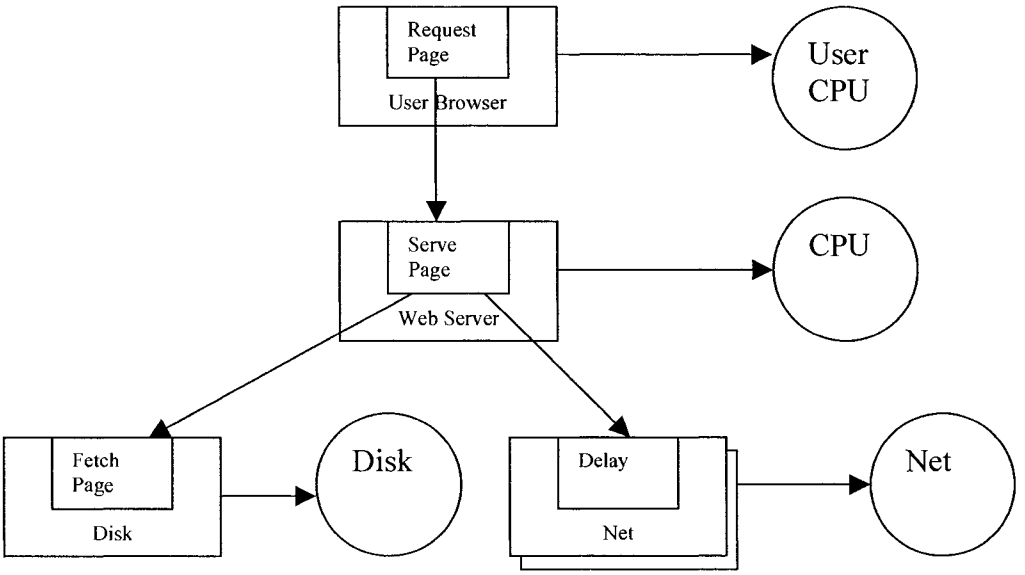

Figure 7: An example of the LQN model [17]

There are two solvers for layered queues, which take the same input format. One uses analytic mean-value queuing approximations to solve the queues at all the entities,

while the other is a simulator. There is also an experiment controller that can execute parameterized experiments over parameter ranges. LQN models can describe the relationship between messages and processes, and clearly shows the processes that use the resources. The toolset can calculate the performance result automatically.

Amyot et al. [31] utilize a visual notation called Use Case Maps (UCMs) for capturing the requirements of reactive and distributed systems. Compared to the Unified Modeling Language (UML), UCMs fit in between Use Cases and UML behavioural diagrams. In UML, Class Diagrams are used to describe how a system is constructed, but do not describe how it works. This task is taken up by UCMs. A Use Case Map can describe one or more scenarios unfolding throughout a system. The UCM Navigator (UCMNav) was developed as an editor and a repository manager, and has been used by industrial associates to create large, industry-scale scenario specifications.

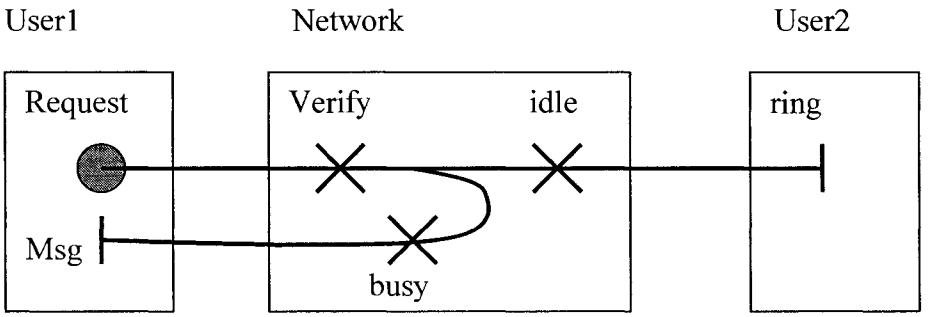

Figure 8: An example of use case maps [31]

Figure 8 shows an example of a use case map. It is composed of a start point (filled circle), three responsibilities (crosses), and two end points (bars). The responsibilities can be bound to components, which are software entities or hardware nodes. The start point is called "request", and it triggers the path. "Verify", "idle" and

"busy" are responsibilities that represent the functions to be performed. "Msg" and "ring" are the end points that represent the result or effect. User1 makes a call to User2 through the network. If the line is idle, User2 is triggered by ringing; otherwise, a feedback message returns to User1. UCMs describe scenarios of a system. They can also show the deployment of the processes in the nodes. The function of each responsibility is clearly marked. Paths of system behaviours are also shown in the diagrams.

Buy et al, [32] present a Petri-Net-based modeling approach for the real-time program analysis. Buy's work extends the Petri Net to a Simple Timed Petri-Net (STP) with the addition of a firing delay on each transition to model several timing properties. For instance, if a transition represents a certain computation, its firing delay can be used to represent the time required by the computation. Figure 9 shows an example of STP modeling a timeout clock. Place p_1 and p_2 represent two locations in processes *P_1* and *P_2*, and transition t_2 represents a communication between the processes. In this case, *P_1* does not wish to wait for longer than 10 time units to communicate with *P_2*. The timeout is suitably captured by transition t_1. If p_2 receives a token before p_1 does or within 10 time units after p_1 does, the t_1 is not fired, and t_2 is fired instead. If p_2 does not receive a token within 10 time units after p_1 receive a token, t_1 fires; thus preventing the firing of t_2. The Simple Timed Petri-Net describes the relationship between processes and messages publishing. It also marks the time delay before the message publishing. The dependency among the processes can be shown clearly.

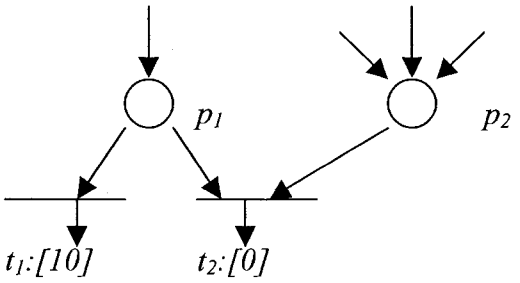

Figure 9: An example of the simple timed Petri-Net modeling a timeout clock [32]

3.2 Environment Overhead Measurement

Liu and Layland [16] introduced Rate monotonic scheduling theory in 1973. Rate monotonic analysis (RMA) is a mathematical approach that helps ensure that a real-time system meets its performance requirements. Five assumptions need to be made for the real-time system to make the schedulability analysis possible, such as neglecting of environment overheads. These assumptions prevent the practical use of RMA.

Katcher et al. [22] relax the assumption of neglecting the environment overheads, such as the cost of preemption, including interrupt handling, scheduling and context swapping, etc. Their works presents a methodology for incorporating the costs of scheduler implementation within the context of fixed priority scheduling algorithms. This work provides a first step toward bridging the gap between the real-time scheduling theory and implementation realities. Katcher et al. define following overheads that happen in the kernel:

C_{int}: the time to handle an interrupt.

C_{sched}: the time to execute the scheduling code to determine the next task to run.

C_{resume}: the time to return to the previously active task when preemption does not occur.

C_{store}: the time to save the state of the active task to a task control block (TCB), and then perform an insertion sort of the TCB into the run queue.

C_{load}: the time to load the new active task state from the run queue.

C_{trap}: the time to handle the trap generated by normal completion of a task.

When an integrated interrupt occurs, in which hardware interrupt priorities are matched with the software task priorities, the total overhead to process an interrupt is $C_{preempt}= C_{int}+C_{sched}+C_{store}+C_{load}$. After a task has completed, the total overhead to clean up the task is $C_{exit}=C_{trap}+C_{load}$. The worst-case overhead for each fixed priority periodic task running under an integrated interrupt event driven scheduling mechanism is $C_{preempt}+C_{exit}$. When a nonintegrated interrupt occurs, in which the priority of the interrupt associated with a task's arrival has no correspondence to the software priority of that task, the total overhead to process an interrupt is still $C_{preempt}= C_{int}+C_{sched}+C_{store}+C_{load}$ in the case of preemption. When no preemption happens, the total overhead to process an interrupt is $C_{nonpreempt}= C_{int}+C_{sched}+C_{resume}$. Without loss of generality, an assumption is made that the sum of $C_{store} + C_{load}$ is greater than C_{resume}, so that $C_{preempt} > C_{nonpreemt}$. All the parts of overheads that are introduced by Katcher et al. need to be measured on the target system.

Stewart [28] provides a measurement method for measuring execution time and real-time performance in his work. A logic analyzer method of measuring the execution time is used as the basis for strategically measuring different parts of the code. For measuring the context switch overhead, an easy way is to build an application of two periodic tasks with different priorities that do nothing more than toggle bits on an 8-bit

digital output port. This is the same port used for collecting data with the logic analyzer. The slowest task (in this case connected to channel 1 of the logic analyzer) would be pre-empted by the higher task (connected to channel 2), yielding a timing diagram as shown in Figure 10. The RTOS overhead Δthr can be approximated as: $\Delta thr = (t_1 - t_2 - t_3)/2$. The reason t_3 is also subtracted from $t_1 - t_2$ is that during the long pulse of t_1, some code of t_1 was executed: the code to produce one pulse, hence the time for t_3. It is possible that the pulse on channel 1 is during the low cycle, which is when channel 1 is 0. In this case, t_3 should be measured as the length of a 0 pulse instead of the length of a 1 pulse. Stewart's method can measure the context switch overhead accurately and precisely.

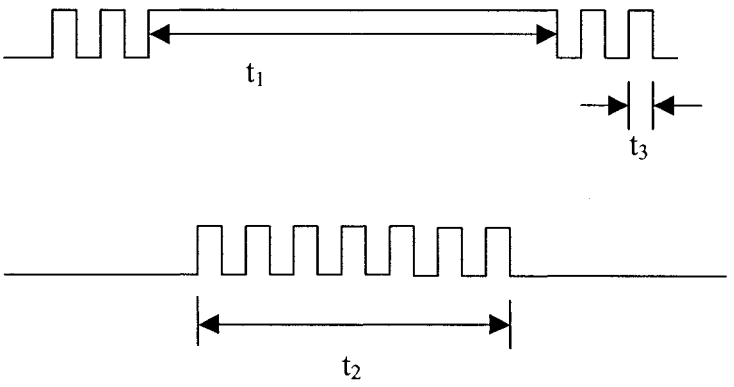

Figure 10: Measuring the context switch overhead [28]

Lai et al, [29] estimates the context switch time by measuring the time to write a byte to another process and then read one byte reply. For more than one process, the byte is passed around in a round-robin fashion through a ring of processes. The overhead of the pipe operations is included in measurement results. McVoy [30] also estimates the context switch time by using a ring of two to twenty processes that are connected with Unix pipes. A token is passed from one process to another process, forcing the context

switches. The benchmark measures the time needed to pass the token two thousand times from process to process. Each transfer of the token has two costs: the context switch, and the overhead of passing the token. The benchmark measures the cost of passing the token through a ring of pipes in a single process to calculate just the context switching time. This overhead time is defined as the cost of passing the token with in a single process and is not included in the context switch time. Lai and McVoy's methods are simply and easy to use on the target system that based on a Unix/Linux operating system.

3.3 Prediction Algorithms

Lehoczky et al, [33] provide a useful algorithm for calculating the worst-case response time of a process that is running with a set of independent periodic processes in a single computer. If a process has the highest priority, its worst-case response time evidently will be equal to its own execution time. For a general process P_i the worst-case response time R_i is given as:

$$R_i = S_i + Dh_i \qquad (3.1)$$

where S_i is the service time of process P_i; Dh_i is the delay caused by any process P_j whose priority is higher than priority of process P_i in any time interval $[t, t + R_i)$. At a worst-case when all higher priority processes are released along with process P_i, the delay Dh_i will be maximum. The total delay caused by higher priority process for P_i is:

$$Dh_i = \sum_{j \in hp(i)} \lceil R_i / Per_j \rceil S_j \qquad (0<R_i<=Per_i) \qquad (3.2)$$

where *hp(i)* is the set of higher-priority processes compared with P_i, and Per_j is the period of process P_j that belongs to the set *hp(i)*. $\lceil R_i / Per_j \rceil$ is the number of times that the P_j is released within the time interval $[0, R_i)$. Combining equation 3.1 and 3.2, the worst-case response time of process Pi can be calculated as following equation 3.3:

$$R_i = S_i + \sum_{j \in hp(i)} \lceil R_i / Per_j \rceil S_j \qquad (3.3)$$

Due to the ceiling functions, it is difficult to solve for R_i. An iterative approach can be used as a solution. Equation 3.3 can be rewritten as following equation 3.4, and the iteration starts with $R_i(0)=S_i$ and terminate when $R_i(n)=R_i(n-1)$.

$$R_i^{n+1} = S_i + \sum_{j \in hp(i)} \lceil R_i^n / Per_j \rceil S_j \qquad (3.4)$$

For example, suppose three processes P_1, P_2 and P_3 run on a single computer. $Per_1=20$, $Per_2=50$, $Per_3=100$, $S_1=5$, $S_2=10$, $S_3=20$.The priority sequence is $P_1>P_2>P_3$. According to Lehoczky's algorithm, the worst-response time of process P_3 can be calculated by the following steps: $R_3(0)=20$; $R_3(1)=35$; $R_3(2)=40$; $R_3(3)=40$. So the worst-response time of process P_3 is *40*.

Liu and Layland define the CPU utilization factor for fixed priority periodic systems to be the fraction of processor time spent in the execution of the process set. In other words, the utilization factor is equal to one minus the fraction of idle processor time. For a process P_i, if S_i is the service time of process P_i, Per_i is the period of process P_i, and there are a total of *m* processes running on given CPU, the CPU utilization factor is:

$$U = \sum_{i=1}^{m} (S_i / Per_i). \qquad (3.5)$$

For example, suppose three processes P_1, P_2 and P_3 run on a single computer. $Per_1=20$, $Per_2=50$, $Per_3=100$, $S_1=5$, $S_2=10$, $S_3=20$. According to Liu and Layland's definition, the CPU utilization can be calculated as following:

$U=5/20+10/50+20/100=65\%$.

An analysis of these related works is provided in the next chapter to indicate the limitations of current modeling approaches if they are used for the performance prediction of a loosely coupled DRE system.

Chapter 4 The Thesis

Even though performance modeling approaches introduced in the previous chapter are well-known and widely used, they still have some limitations that do not match the needs of the GeoSurv II UAV project. An analysis is provided in Section 4.1 to show the limitations. Section 4.2 provides an ideal solution that overcomes the limitations associated with the visual notation, environment overheads measurement, and prediction algorithms. Section 4.3 states the objective in terms of a proposed solution, called Path Network modeling. The contributions to the visual notation, environment overheads measurement, and prediction algorithms are outlined in Section 4.4, and Section 4.5 provides limitations and assumptions that underlie the research.

4.1 Current Research Limitations

The target system for the performance prediction is a small-scale loosely coupled DRE system, such as that used in GeoSurv II UAV project. An analysis is presented here to show limitations of the current approaches as applied to the target system. This analysis focuses on visual notation, environment overhead measurement, and prediction algorithms.

4.1.1 Limitations for Visual Notation

The goal of the visual notation is to provide a graphic performance model of the target system. The graphic model is desirable as it shows all the structure elements and the temporal parameters in the same diagram. Application developers in the UAV team can easily abstract the structural attributes and collect the service time of each process from the diagram to calculate the scenario response time and the CPU utilization. Existing visual notations address the needs of performance prediction to varying degrees. However, no existing visual notation meets all the needs of the GeoSurv II UAV project.

A diagram showing the message paths and nodes is desirable to show where the network communication overheads occur along a given message path. This is because the network communication overheads are needed to take into account in the prediction of the scenario response time. Live Sequence Charts and UML sequence diagrams do not show each node and do not indicate if a message path is a distributed (between two nodes) or local (with in a node) path within one diagram.

Different icons to represent application processing and environment overheads, such as publishing overheads, subscribing overheads and daemon service overheads, are desirable to explicitly show the various sources of processing demanded in a scenario. These icons also help application developers to collect the service time of each process directly from the diagram. Live Sequence Charts, UML sequence diagrams, task graphs, LQN, Use Case Maps (UCMs) and Petri-Nets do not show these icons. The lifeline in a

sequence diagram and the delay in a timed Petri-Net only indicate the total service time of a process and do not break it down into constituent components.

Message path lines with message IDs that represent each message are desired to show the direction and the path of each message in the diagram. This is because message path lines can help application developers determine the trace of each scenario from the diagram. Task graphs and UCMs do not have message path lines. The paths of UCMs connecting the different responsibilities only indicate the dependency and running sequence of these responsibilities. An arrow in a task graph only represents the output and input between two processes. Message IDs are not show on the either of these types of diagrams. Without a message icon and ID, the message publishing sequence in a scenario may be difficult to identify in a diagram.

It is important to show the dependent relationship among messages in the diagram because each scenario consists of a chain of message paths that have dependent relationship. Application developers can directly find out the trace of each scenario according to the message's dependent relationship from the diagram. Task graphs and the RT-PSC model do not show this relationship. This leads to difficulty in identifying each scenario without referencing other documents.

The location of the middleware daemon needs to be identified because it will help application developers to take all daemon overheads into account when they calculate the performance. None of the current visual notations shows the daemon location. Even though the daemon is actually a process running on a node, the special role of the daemon in publish/subscribe middleware makes it important to distinguish the daemon from application processes.

The marking of process priority and message priority is desirable in the diagram because the priorities of processes and messages have key roles in deciding the process running sequence and the message publishing sequence, and these sequences may influence the prediction of the scenario response time. None of the current visual notations mark these priorities. This leads to confusion in understanding the process running sequence and the message publishing sequence when the higher priority pre-emption happens.

4.1.2 Limitations for Environment Overhead Measurement

The different kinds of environment overheads of a DRE system need to be estimated or measured in a practical way. Katcher et al. relax the assumption of Liu and Layland for neglecting the environment overhead, and all the parts of overheads that are described by Katcher et al. need to be measured on the target system. However, without intimate knowledge of the operating system kernel, it is hard to measure the overhead using Katcher's method. Stewart provides a method for measuring context switch overhead using a logic analyzer, but this increases both the cost and skills needed to make performance measurements. Lai et al. and McVoy estimate the context switch time using Unix pipes, and when combined with the use of hardware timers, the context switch overhead can be measured easily. However, the context switch overhead is only one part of environment overheads. Unfortunately, there is no current work that breaks down the environment overhead for a DRE system based on the CMU-IPC middleware. Methods

for obtaining the different parts of the related overhead, such as publishing overhead, subscribing overhead and daemon service overhead, have not yet appeared.

4.1.3 Limitations for Prediction Algorithms

To predict the scenario response time of a loosely coupled DRE system, the different kinds of environment overheads should be considered and the delay caused by the middleware daemon and other higher priority processes also should be taken into account. Even though Lehoczky et al. accounted for the delays caused by the higher priority processes, the delay caused by middleware daemon still needs to be included in the calculation. Lehoczky treats each process as a single-period independent process that is always executed in a fixed period, but processes in the target system may be multi-period processes. A multi-period process can subscribe to several messages with different publishing rates, so it is executed in several fixed periods. Such multi-period processes should also be dealt with in the calculation of the scenario response time.

Both the processing time and environment overheads should be considered when calculating the CPU utilization of each node. Liu and Layland assume that environment overhead can be neglected. Even though Katcher et al. have considered the influence of context switch overhead, other environment overheads, such as publishing overheads, subscribing overheads and daemon service overheads, have not been considered yet in their method. The lack of environment overheads makes the prediction of the CPU utilization less accurate.

An analysis tool helps to reduce the calculation burden in predicting a large-scale DRE system, but it also adds the burden of learning the tool. For a small-scale DRE system, a simply and easy to learn prediction algorithm that can be used manually is desirable to allow application developers to calculate the performance. Live Sequence Charts, UML sequence diagrams, task graphs and Use Case Maps (UCMs) are only modeling approaches for applications, and they do not come with performance analysis methods or tools. The LQN modeling has tools that use analytic mean-value queuing approximations to solve the queues of all the entities. Petri-Net and its extensions also come with a vast number of analysis tools that are based on the formal methods and notations. To master these tools a deep knowledge of performance analysis, operating system and mathematics is needed. This will increase the time and difficulty for application developers. A simple prediction algorithm is needed that application developers can learn and use easily.

4.2 An Ideal Solution

Currently, despite the many approaches for performance modeling and prediction that are in use, there is no particular method for the performance prediction of a small-scale loosely coupled DRE system that is based on the CMU-IPC middleware. An ideal solution should absorb the advantages of the existing modeling approaches, while avoiding the drawbacks, and introduce appropriate new features as necessary.

Perhaps an ideal modeling approach should include a visual notation that can clearly show the elements and parameters listed below and their interactions in the same diagram:

- The processes, messages, nodes and daemon.
- The process priority and message priority.
- The process processing time for each message.
- The message sending and receiving overheads.
- The middleware daemon service overhead.
- The process blocking overhead.
- The network communication overhead.
- The context switch overhead.
- The dependency among messages.

An ideal modeling approach should also break down the environment overhead into the following pieces:

- The context switch overhead.
- The message sending overhead.
- The message receiving overhead.
- The daemon servicing overhead.
- The process blocking overhead.
- The network communication overhead.

An ideal modeling approach should also include prediction algorithms that have following advantages:

- Consider multi-period processes.
- Consider the delay caused by higher priority process.
- Consider the delay caused by middleware daemon.
- Do not need special software tools and expensive instruments.
- Do not need deep knowledge of operating system kernel.
- Do not need deep knowledge of performance analysis and mathematics.
- Easy to calculate performance results manually.
- Have a fast learning curve.

An ideal modeling methodology should separate the concerns of application developers and system designers when using the modeling approach. System designers, who know the system environment well, can measure the environment overheads of the target system. Application developers, who are familiar with the functional requirement of the UAV system, can use the visual notation to build a graphic model of the target system. Then application developers can use the following as the inputs to the prediction algorithms to calculate the performance:

- Values of application processing time measured by application developers,
- Values of environment overheads obtained from system designers,
- Values of structural attributes obtained from a graphic model of the target system,

The separation of the concerns of application developers and system designers allows application developers to gain valuable information about the performance of their applications in the target system.

4.3 The Statement of the Thesis

Determining the performance of the GeoSurv II application is a challenging aspect of development. Path Network (PN) modeling is proposed to meet this challenge. The visual PN notation emphasizes the aspects of the application and its DRE environment that contribute to application performance. Algorithms are presented for measuring temporal properties of the environment and calculating the scenario response times and CPU utilization. The algorithms are designed to separate the concerns of application developers from those of system designers. This separation allows application developers, who may have less knowledge of real-time systems, to gain valuable information about the performance of the GeoSurv II UAV application. Furthermore, this performance information will be essential input to the design of the on-board DRE architecture.

4.4 The Contribution of the Thesis

The thesis contributes knowledge in the following areas:

(1) Visual notation: the visual notation of the PN model is designed to capture the relevant aspects of the GeoSurv II UAV application, its network architecture and deployment that are needed for the GeoSurv II application performance analysis.

(2) Environment overheads measurement: the PN model identifies the relevant aspects of the environment overhead encompassing the CMU-IPC middleware,

the operating system and network communication, and gives algorithms for measuring each aspect of the environment overheads.

(3) Application performance prediction: algorithms are presented for calculating the scenario response times and CPU utilization of an application represented as a PN model.

4.5 The Scope of the Thesis

PN modeling provides a general guideline, rather than a strict upper bound, to help application developers gain more confidence on the performance of the UAV application. The prediction is made according to a practical rule that is based on a pessimistic formal model and potentially inexact measurements. The prediction result is not intended to be strictly accurate or reliable for every situation. There is no guarantee that the prediction results will be an absolutely safe least upper bound of the performance. PN modeling is an easily learned and easily applied procedure for calculating an approximation of the performance the UAV application.

Certain assumptions described below must be made about the target system to obtain the prediction results for the UAV application on the target system, and the effort of relaxing these assumptions is left for future research.

Assumption 1: Each node is limited to contain only one mono-core CPU.

Assumption 2: There is only one middleware daemon in the whole system. The daemon has the highest priority among all processes.

Assumption 3: The message publishing and subscribing happens among the processes only, and threads are not considered separately.

Assumption 4: The messages in the system only use primitive data types. The sizes of the messages are constant and not a variable of the model input.

Assumption 5: The target system is built on event-driven (or triggered) architecture not time-driven architecture. A message can trigger one or more succeeding messages, but it can only be triggered by one preceding message.

Assumption 6: Each message has a fixed publishing rate and period, and all the messages in a scenario have the same period. The deadline of a scenario does not exceed its period.

Assumption 7: All processes and the daemon block themselves after finishing their jobs.

As initial research, PN modeling is the first step in predicting the performance of the GeoSurv II UAV on-board DRE system. Until now, both the development of the GeoSurv II UAV autonomy software and the performance analysis of the on-board computing architecture use the experimental system with non-real-time components such as the IPC middleware, the Linux operating system and the Ethernet network system. The final on-board computing architecture has not been decided yet. As a simulation environment, this experimental system served as a test-bed that is compatible with the on-board DRE system. Compared with the non-real-time experimental system, the final real-time architecture is believed having low and stable latency. This will lead to more accurate prediction results using PN modeling on the final system. Future research will be

based on a publish/subscribe middleware with improved real-time capabilities and a more deterministic platform.

4.6 Organization of the Dissertation

The reminder of the document is organized as three chapters. Chapter 5 describes Path Network modeling. An outline of PN modeling is introduced in Section 5.1. The visual notation of PN modeling is described in Section 5.2. Application developers can use the visual notation to build a PN model for the UAV application. A full description of an environment overhead model for a loosely coupled DRE system based on the CMU-IPC publish/subscribe middleware is provided in Section 5.3, and the methods of measuring each part of environment overheads are also included. System designers can use these methods to measure the value of the overhead parameters from the target computers. Prediction algorithms are given in Section 5.4 to calculate the two key performance metrics: the scenario response time and the CPU utilization. Application developers can use the algorithms to obtain performance information about their applications. The methodology of PN modeling is summarized in Section 5.5 to clearly assign the roles of system designers and application developers using PN modeling.

Chapter 6 is a case study that focuses on performance prediction for the GeoSurv II UAV autonomy application. A PN model of the UAV Evasive Manoeuvring Mode is developed in Section 6.1, and the structural attributes can be abstracted from the scenario diagram of the PN model. Then, the values of environment overheads of a prototypical test-bed are measured and calculated in the Section 6.2. All the structural attributes,

environment overheads and processing times act as the inputs of the prediction algorithms, and the outputs of the algorithms are the performance prediction results. The steps of the calculation are provided in Section 6.3. Finally, in Section 6.4, the results of the case study are analyzed from three views: the visual notation, the environment overhead measurement, and the performance algorithms.

Chapter 7 gives the conclusions of this research and some suggestions for future research.

Appendix A provides the measurement data and samples of calculation steps for the temporal parameters in the case study.

Chapter 5 Path Network Modeling

Path Network (PN) modeling is an approach to performance modeling and analysis in which an application is represented as a network of message-paths that are evaluated analytically. A PN model is the repository of all relevant structural elements and their relationships, as well as their timing properties such as the processing time and environment overheads. An outline of PN modeling is introduced in Section 5.1. How to use the visual notation of PN modeling to build a PN model is described in Section 5.2. How to measure the environment overheads is provided in Section 5.3. Section 5.4 provides the prediction algorithms for application developers to calculate the performance results. Finally, the methodology of PN modeling is summarized in the last section.

5.1 PN Modeling Guidelines

As a performance modeling and analysis approach, PN modeling includes three parts: using the visual notation to model the application running as a DRE system, measuring the environment overheads and calculating the performance metrics using the prediction algorithms.

The visual notation is used by application developers to build a PN model of a DRE application. The notation provides icons to represent the application and network structure and the environment overheads. The application and network structure are

modeled using processes, messages and nodes. Application developers design the processes and messages based on the GeoSurv II UAV functional requirements. The design is captured in one or more application diagrams, a deployment diagram and one or more scenario diagrams. The application diagram shows the application processes and the message flows among the processes. The diagram focuses on the relationships among processes and messages and is not concerned with deployment to nodes in the target system. The deployment diagram shows the distribution of processes on the nodes of a target system, but does not show message flows. The location of the middleware daemon is an important component in the deployment diagram. A scenario diagram combines message flow information from the application diagram with the distribution information from the deployment diagram, and shows icons representing the environment overheads associated with sending those messages in that particular deployment. The information from scenario diagrams is a key input to the performance prediction algorithms.

The second part of PN modeling involves measuring the environment overheads of the target system. The environment overheads encompass the total performance cost of the DRE environment including the time required for process switching, message publishing and subscribing, daemon execution and network communication. The environment overheads can be broken down according to the source of the overheads. The context switch overhead is a property of the operating system. The application processes experience overheads and blocking associated with sending and receiving messages. The network communication overhead occurs when messages are transmitted between two nodes. The daemon service overhead is inherent to the middleware daemon, and can be broken down into receiving, sending, ACK and blocking overheads. Before

application programmers can apply the prediction algorithms, system designers must provide measurements of the overhead values for the target DRE system.

The third part of PN modeling is the use of the prediction algorithms to calculate the CPU utilization and the response time of scenarios. Application processing time, the environment overheads and the structural attributes are all inputs to the prediction algorithms. Multi-period processes and different kinds of delays are considered in the algorithms. These algorithms can be applied by application developers without the need for special software tools, expensive instruments, deep knowledge of the operating system and hardware, or deep knowledge of performance analysis. Some ideas in calculating the scenario response time are borrowed from Lehoczky et al. [33], such as the use of ceiling function. The calculation of CPU utilization extends Liu and Layland's work [16] to consider environment overheads. Application developers can obtain valuable performance information about their GeoSurv II UAV applications from the performance prediction results and also gain confidence in the safety of their application and the UAV system. The performance information can also provide confidence to system designers while adjusting the design of the on-board DRE architecture.

5.2 Visual Notation

The visual notation of PN modeling helps application developers obtain a graphic view of the performance of a DRE system. The visual notation captures the relevant aspects of the GeoSurv II UAV application, its network architecture and deployments that are needed for the GeoSurv II UAV application performance analysis. The main

structural elements of the DRE system are process, message and node. The relationships among processes, messages and nodes are the main concern of the visual notation. The priorities of processes and messages are explicitly marked on the scenario diagram to help application developers determine the message publishing sequence easily. Different kinds of environment overheads and processing times for each relative message are also shown as easily identified icons on the diagram. The dependencies among messages are clearly shown and application developers can easily find out every preceding and succeeding messages of a given message. Application diagrams, a deployment diagram and scenario diagrams are drawn to build a PN model of an application.

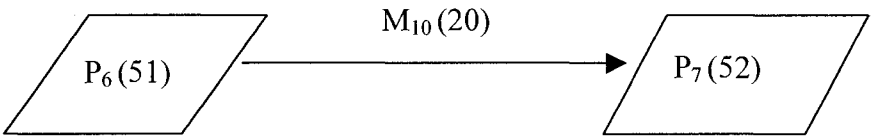

Figure 11: An example of an application diagram

Application diagrams for each mode of the application are drawn first to show the publish/subscribe relationships between processes and messages to help application developers abstract the application structure in a graphical way. Figure 11 shows an example of an application diagram with two processes named P_6 and P_7 and one message named M_{10}. The processes are represented by parallelograms with their process IDs and priorities, and the message is shown as a directed line with its message ID and its priority. P_6 publishes M_{10} that is subscribed by P_7. The priority of P_6 is marked as 51, and the priority of P_7 is marked as 52. The message M_{10}'s priority is 20. For all messages and

processes, the higher the priority, the larger the value of the priority. The process P_7 has a higher priority than P_6 does. Higher priorities are scheduled before lower priorities.

The deployment diagram focuses on the distribution of the processes on the nodes of a distributed system. Figure 12 shows an example of a deployment diagram. All the processes in Figure 11 are deployed in Figure 12. There are only two nodes named $Node_1$ and $Node_2$ in the system. The nodes are represented by dashed-line rectangles with their node IDs. Each node can contain several processes. P_6 is deployed on $Node_1$, and P_7 is deployed on $Node_2$. The middleware daemon is also deployed on the $Node_1$.It is represented by a dashed-line circle with letters "DM" and the priority (88).

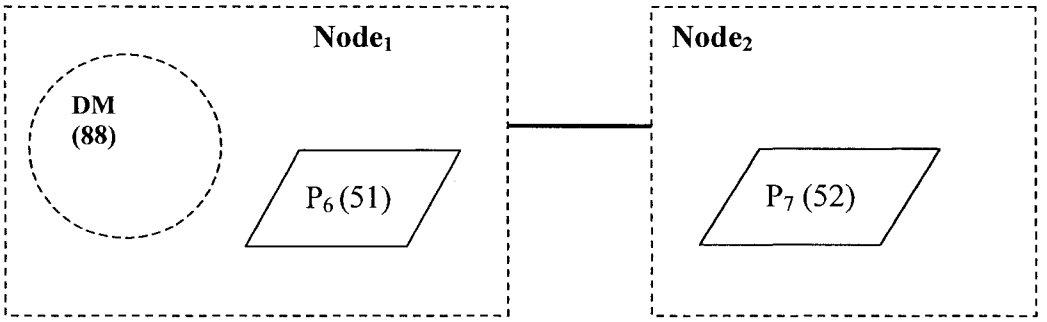

Figure 12: An example of a deployment diagram

The most important diagram in a PN model is the scenario diagram that provides the structural attributes and icons of temporal parameters for the performance prediction. An example of a scenario diagram is shown in Figure 13. The scenario diagram in Figure 13 is a combination of the application diagram in Figure 11 and the deployment diagram in Figure 12. A message is represented by a directed arrow as in the application diagram, and it is identified and given a priority. The following icons are explained in a sequence from left to right along with the scenario. A message start point, also called a message

trigger point, is a circle with message sequence number inside, and is located in P_6. A scenario ID, Sc_2, is labelled near the first message start point of the scenario. The context switch overheads are represented by sun like icons, and are located at the time points when processes obtain their turns to run. The processing times are represented by circles with a letter 'T' inside, and are located in both P_6 and P_7. A message sending overhead is represented by a circle with a letter 'S' inside, and is located in P_6. The process blocking overheads are represented by circles with a letter 'B' inside, and are located in both P_6 and P_7. The daemon service overhead is represented by a diamond, and is located outside of P_6 in $Node_1$. The context with overheads occurred when the daemon obtain its turns to run are included in the daemon service overhead and are not marked in the scenario diagram. The network communication overhead is represented by a hollow circle, and is located on the message path between two nodes. A message receiving overhead is represented by a circle with a letter 'R' inside, and is located in P_7. A message end point is a bar, and is located in P_7.

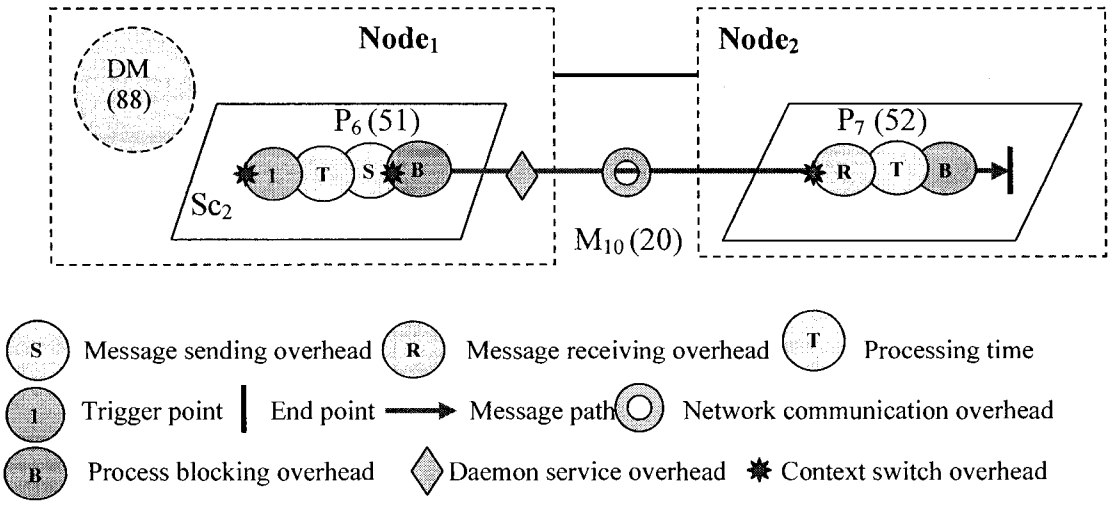

Figure 13: An example of a scenario diagram

To mark the context switch overhead icons on a scenario diagram, application developers must find out the time points of process switching on each node according to the process priorities and message priorities. In the example of Figure 13, the context switch overheads occur in following 3 time points from left to right along Sc_2:

- When P_6 is triggered by an internal timer and begins to run,
- When P_6 obtains its turn to self-block after pre-empted by the Daemon,
- When P_7 is triggered by message M_{10}.

Device processes are special cases of processes like P_6 and P_7 in Figure 13. Device processes can be divided into source processes and sink processes. A source process, such as P_6, publishs at least one message that is triggered by an internal timer or other sensor device. A sink process, such as P_7, subscribes to at least one message that is used only to control an outside actuator. A scenario starts from one source process and ends at one or more sink processes.

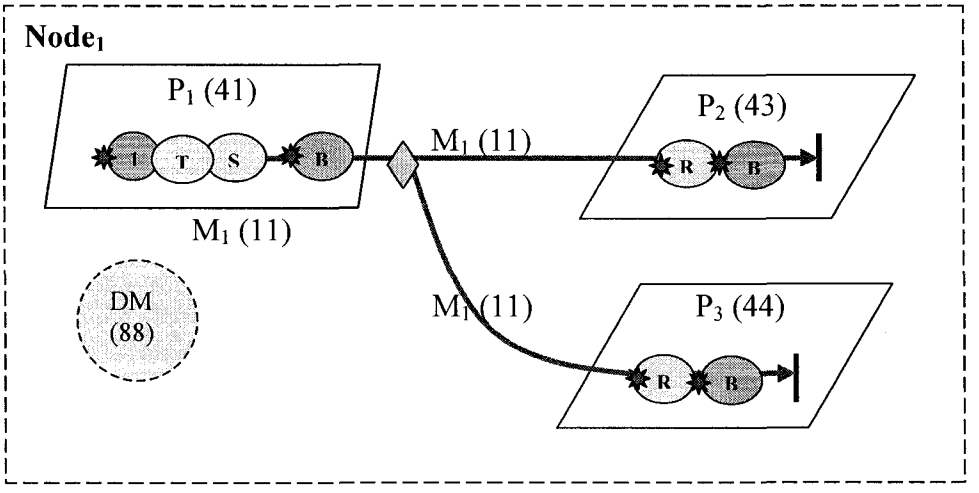

Figure 14: A sample of complex message path structure

There are two kinds of message paths: simple message paths and complex message paths. A simple message path only has one process that subscribes to the message. The message path shown in Figure 13 is a simple message path. A complex message path has two or more processes that subscribe to the same message. Figure 14 shows a complex message path. P_2 and P_3 subscribe to the same message M_1 from P_1.

Processes can be divided into two categories: single-period processes and multi-period processes. A single-period process only subscribes to one message, so the process has only one period that is same as the period of the message. A single-period process only belongs to one scenario, and its period is also the same as the period of its scenario. Figure 14 shows a single period scenario in which all of the processes have the same period determined by the publication of M_1. A multi-period process can subscribe to two or more messages, and each message may have different publishing periods. A multi-period process belongs to two or more scenarios, and the process runs in different periods to match the relevant scenarios. Figure 15 shows an example of a multi-period process in a scenario diagram. P_2 is a multi-period process, and the other processes are single-period processes. The scenario Sc_1 contains two messages M_1 and M_2, and the scenario Sc_2 contains two messages M_3 and M_4. P_2 belongs to both Sc_1 and Sc_2. The period of Sc_1 may be different from the period of Sc_2. For example, if the periods of Sc_1 and Sc_2 are 100ms and 50 ms, P_2 is triggered by two messages every 100 ms and 50 ms.

A message publishing sequence number is marked at the start point of each message path within its own scenario according to the trigger time and the priorities of processes and messages. For example, in the scenario Sc_1 of Figure 15, M_1 is published prior to M_2. The start point of M_1 is marked as 1, and the start point of M_2 is marked as 2.

In the scenario Sc_2, M_3 is published prior to M_4, and the start point of M_3 is marked as 1. Both messages M_1 and M_3 are published first in their own scenarios. However, the process release time and process priority decide the sequence that messages M_1 and M_3 are published within the whole application.

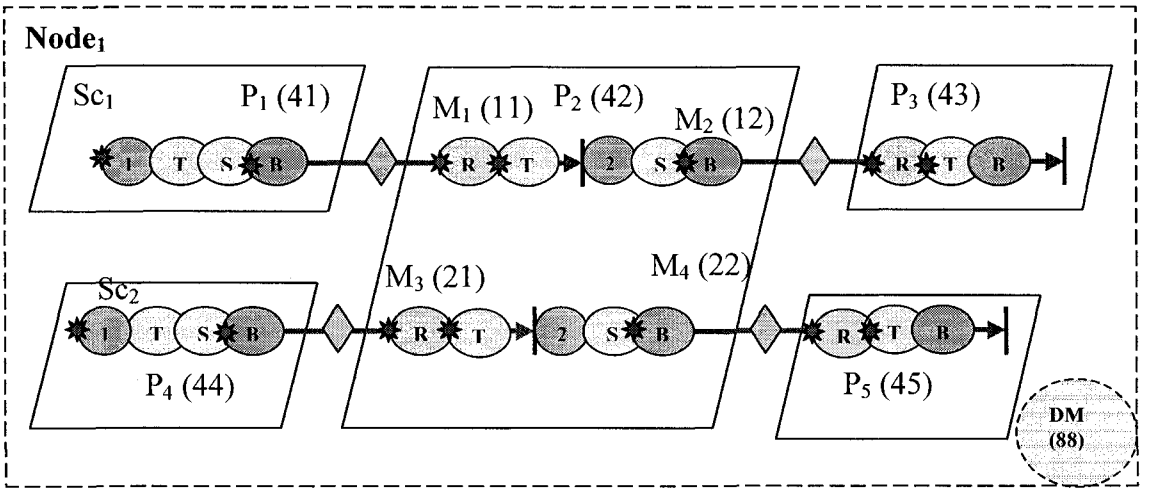

Figure 15: An example of a multi-period process

5.3 Environment Overhead Measurement

Methods to measure the different kinds of environment overheads represented in scenario diagrams are a part of PN modeling. The measurement methods presented here are designed for DRE systems that utilize the CMU-IPC publish/subscribe middleware. If the UAV team decides to use a new publish/subscribe middleware in the future, the methods should be changed in accordance with the architecture of the new middleware.

The remainder of Section 5.3 is broken down as follows:

The definitions of different parts of the environment overheads and the process service time are listed in Section 5.3.1. The method to measure context switch overhead is given in Section 5.3.2. The methods to measure the environment overheads that occur inside the processes and daemon are given in Section 5.3.3. Finally, the method to measure the network communication overhead is given in Section 5.3.4.

5.3.1 Environment Overhead Model

Environment overheads encompass the total performance cost of the DRE environment. As shown in Figure 3 in Section 2.2, the DRE environment encapsulates a publish/subscribe middleware, an operating system and the DRE hardware. Environment overheads can be broken down into several parts according to the source of overheads. An example of an environment overhead model for a scenario is shown in Figure 16. It also shows the mapping from the environment overhead model to the scenario diagram. The definitions of each part of the environment overheads are listed below and are followed by a detailed explanation of the environment overhead model and mapping shown in Figure 16.

(1) The message sending overhead, Es_j, is the time interval for a process in $Node_j$ to send a message to the daemon. For example, Es_1 is a message sending overhead that occurs in $Node_1$.

(2) The message receiving overhead, Er_j, is the time interval for a process in $Node_j$ to receive a message from the daemon and sending back an ACK. For example, Er_2 is a message receiving overhead that occurs in $Node_2$.

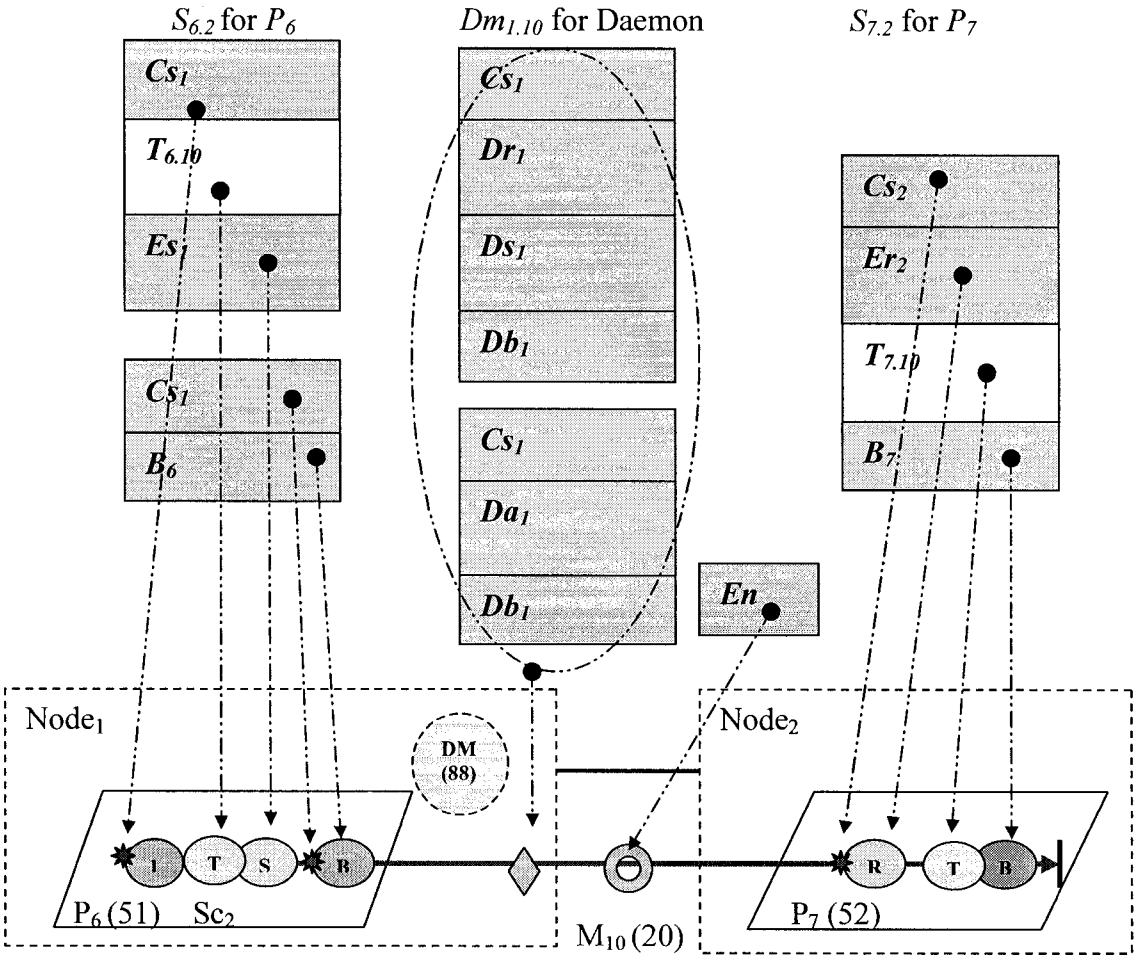

Figure 16: Breakdown of processing for a scenario

(3) The daemon receiving overhead, Dr_j, is the time interval for the daemon in $Node_j$ to receive a message from a process. For example, Dr_1 is a daemon receiving overhead that occurs in $Node_1$.

(4) The daemon sending overhead, Ds_j, is the time interval for the daemon in $Node_j$ to send a message to a process. For example, Ds_1 is a daemon sending overhead that occurs in $Node_1$.

(5) The daemon obtaining ACK overhead, Da_j, is the time interval for the daemon in $Node_j$ to obtain an ACK from a process. For example, Da_1 is a daemon obtaining ACK overhead that occurs in $Node_1$.

(6) The daemon blocking overhead, Db_j, is the time interval needed for the daemon to self-block. For example, Db_1 is a daemon blocking overhead that occurs in $Node_1$.

(7) The process P_i blocking overhead, B_i, is the time interval needed for the process P_i to self-block. For example, B_6 is a process blocking overhead that occurs in P_6.

(8) The network communication overhead, *En*, is the time interval from a message leaving one node to the message arriving at another node.

(9) The context switch overhead, Cs_j, is the overhead of context switching and task scheduling that occurs before process execution in $Node_j$. For example, Cs_1 is a context switch overhead that occurs in $Node_1$.

The scenario diagram reflects all the different environment overheads and the processing time on message paths as shown in Figure 16. The grey blocks represent the different kinds of environment overheads, and the white blocks represent the application processing time of each process. The first and third columns of blocks represent the process service times, while the middle column represents daemon and network overheads. The process service time $S_{i,n}$ is defined as the CPU time cost of process P_i in the scenario Sc_n. For example, $S_{6,2}$ is the process service time of P_6 in Sc_2, and $S_{7,2}$ is the process service time of P_7 in Sc_2.

The process service time not only includes different kinds of environment overheads, but also includes the processing time for the preparing or handling of messages in each process. The processing time of message M_k by process P_i is noted as

$T_{i,k}$. For example, $T_{6,10}$ represents the processing time for P_6 preparing the message M_{10}, and $T_{7,10}$ represents the processing time for P_7 handling the message M_{10}. It must be noted that a process can prepare or handle one or more messages. According to the scenario diagram, a process service time can be calculated by adding together all the processing times and overheads that are included in the given process. For example, the process service time $S_{6,2}$ can be determined from the scenario diagram as follows:

$$S_{6,2} = (Cs_1+T_{6,10}+ Es_1)+(Cs_1+B_6).$$

The daemon service overhead $Dm_{v,k}$ is the CPU time cost on the daemon for a given message M_k when the daemon is located in $Node_v$. For example, The second column of blocks in the Figure 16 represents the daemon service overhead $Dm_{1,10}$ for the message M_{10}. The daemon service overhead $Dm_{1,10}$ can be calculated from the second column of the overhead model as:

$$Dm_{1,10}= (Cs_1+Dr_1+Ds_1+Db_1)+(Cs_1+Da_1+Db_1).$$

The environment overhead model not only has an important role in measuring environment overheads as shown in the remainder of this section, but it also helps application developers to understand the prediction algorithms. In some simple cases the prediction result can be calculated directly from a scenario diagram that accurately reflects the environment overhead model. For example, if application developers want to calculate the scenario Sc_2's response time Rsc_2 in Figure 16, the only calculation required is adding up all the time intervals for the scenario $Sc2$ as: $Rsc_2=S_{6,2}+Dm_{1,10}+En+S_{7,2}$. In more general cases, the delay caused by higher priority processes and the parallel execution of processes of a scenario in different nodes also must be considered. The

calculation can be more complex compared with the example above. The prediction algorithms for the generic cases are described in Section 5.4.

5.3.2 Measuring Context Switch Overhead (Test-Set-01)

The method used to measure the context switch overhead comes from Lai's work. The method is used for the Unix/Linux operating systems, and is performed by reading and writing to a Unix/Linux pipe separately from two processes. If the UAV team decides to use an operating system that does not support pipes in the final architecture, this method must be changed.

Figure 17 shows a relative timeline of the activities involved in each iteration of the test. Two processes, referred to as the parent process and child process, are needed to measure the process context switch overhead of a given node. The parent process is set to a higher priority and the child process is set to a lower priority. The parent process contains 100 loops of the system call that reads from a Unix/Linux pipe. The child process contains 100 loops of the system call that writes to a Unix/Linux pipe. Timestamps are generated before and after the system writing call in the child process. The latency, which is noted as *La,* is measured from the start timestamp to the end timestamp. The executing sequence of these two processes is as follows:

(1) Parent runs first, is initially blocked by reading the pipe, and switches to Child.

(2) Child runs and obtains the start timestamp.

(3) Child writes to the pipe.

(4) Context switches from Child to Parent.

(5) Parent reads a byte from the pipe, loops to read the next byte, and is blocked again.

(6) Context switches from Parent to Child.

(7) Child obtains the end timestamp.

(8) Repeat from step 2 for 99 more times

The time interval from the start timestamp to the end timestamp includes 2 context switch overheads, 1 blocking pipe reading time Pr_b and 1 pipe writing time Pw. The value of the context switch overhead Cs can be calculated using equation 5.1:

$$Cs=(La - Pr_b - Pw) / 2 \qquad (5.1)$$

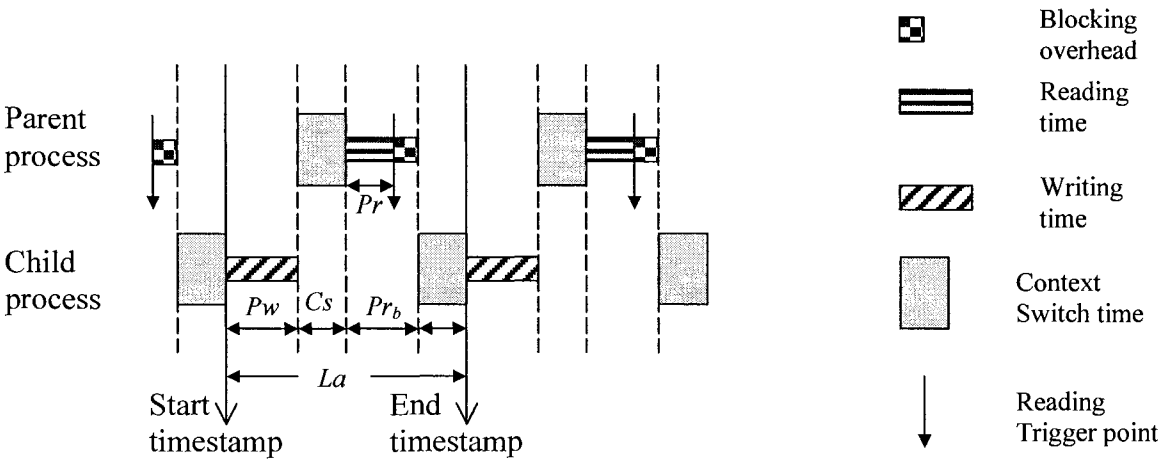

Figure 17: Context switching between two processes for pipe reading and writing

The difference between the blocking pipe reading time Pr_b and the non-blocking pipe reading time Pr is the blocking overhead, as shown in Figure 17. The blocking overhead is used to determine whether the pipe size is 0. Because the blocking overhead is hard to measure, and much smaller than the context switch overhead, the non-blocking pipe reading time Pr is used to instead of Pr_b in equation 5.1. A single process is used to

measure the pipe reading time *Pr* and the pipe writing time *Pw*. The process writes a byte to the pipe and then reads it from the pipe repeating 100 times. Timestamps are inserted into the process before and after the system *read ()* call to obtain *Pr*, and timestamps are inserted into the process before and after the system and *write ()* call to obtain *Pw*. The test-set for measuring context switch overhead is named as Test-Set-01.

5.3.3 Measuring Message-Related Overheads (Test-Set-02)

To measure and calculate each part of the message-related overheads that occur inside processes and the daemon, the processes P_1 and P_2 are built and run with a middleware daemon on $Node_1$ as shown in Figure 18. P_1 prepares and publishes the message M_1, and P_2 subscribes to the message M_1. Because all processes and the daemon are running on a single node, network communication overhead is not involved. Context switch overhead Cs_1 and processing time $T_{1.1}$ must be measured before calculating the message-related overheads. Cs_1 can be measured by using Test-Set-01 as described earlier. Processing time, such as $T_{1.1}$, can be measured by recording timestamps at the beginning and end of the processing code. The difference between the two timestamps is the value of the processing time.

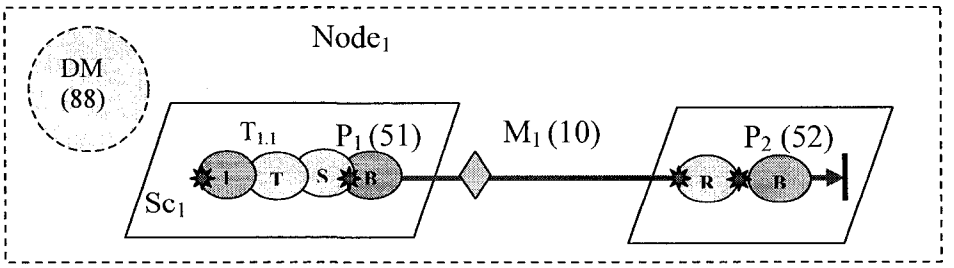

Figure 18: A scenario diagram for measuring environment overhead

The message-related overheads that occur in each process and the daemon can be calculated using the measurement results of four test cases that are shown in Figures 19 through 22. The main differences among these test cases are the variations in the ordering of events that results from the different process priorities. The priorities determine the execution sequence of the processes and the daemon. Note that in two of these tests, the priority of the daemon is lower than that of the application processes. This is contrary to the assumption that the daemon has the highest priority in the GeoSurv II UAV application and has been done here only to generate appropriate data to calculate the message-related overheads. There are five timestamps used in each test case, two of which are instrumented into the middleware daemon. One of these two timestamps is inserted at the point after the daemon obtains a message from a publisher, and the other timestamp is inserted after the daemon sends the message to a subscriber and obtains the ACK from that subscriber. The points at which timestamps are recorded in each test case are marked by the numbered circles in the figures. The latencies between every two adjacent timestamps are measured, as summarized in Table 2.

The process execution sequence can be determined from the priority sequence of each test case. For example, in Figure 19, P_1 has the highest priority 98, so neither the other process nor the daemon can preempt P_1. The daemon, whose priority is 88, can only execute after P_1 finishes publishing message M_1 and self-blocks. The daemon receives M_1, sends it to P_2 and self-blocks. The process P_2, which has the lowest priority 48, begins to receive M_1 and sends back an ACK. P_2 cannot continue running because the daemon is woken by the ACK, and the daemon has a higher priority than P_2 does. The daemon obtains the ACK and then self-blocks again. Finally, P_2 obtains the right to run

the rest of its code and then self-blocks. Now all processes and the daemon are blocked until the next period. Figure 20, 21 and 22 differ only in the ordering of events caused by the different process and daemon priorities. The latency between two timestamps can be easily broken down into the different parts of the processing times and overheads along each curve in Figure 19 through 22, as shown in Equations 5.2 through 5.5.

Case ID	Priority Sequence	Latency from $time_1$ to $time_2$	Latency from $time_2$ to $time_3$	Latency from $time_3$ to $time_4$	Latency from $time_4$ to $time_5$
A	$P_2<DM<P_1$	La_1	La_2	La_3	La_4
B	$P_1<DM<P_2$	Lb_1	Lb_2	Lb_3	Lb_4
C	$P_2<P_1<DM$	Lc_1	Lc_2	Lc_3	Lc_4
D	$P_1<P_2<DM$	Ld_1	Ld_2	Ld_3	Ld_4

Table 2: Latencies need to be measured in Test-Set-02 for a given node

Case A:
$$La_1 = T_{1.1} + Es_1 \quad (5.2)$$
$$La_2 = B_1 + Cs_1 + Dr_1$$
$$La_3 = Ds_1 + Db_1 + Cs_1 + Er_1 + Cs_1 + Da_1$$
$$La_4 = Db_1 + Cs_1$$

Case B:
$$Lb_1 = T_{1.1} + Es_1 + Cs_1 + Dr_1 \quad (5.3)$$
$$Lb_2 = Ds_1 + Cs_1 + Er_1$$
$$Lb_3 = B_2 + Cs_1 + Db_1 + Da_1$$
$$Lb_4 = Db_1 + Cs_1$$

Case C:
$$Lc_1 = T_{1.1} + Es_1 + Cs_1 + Dr_1 \quad (5.4)$$
$$Lc_2 = Ds_1 + Db_1 + Cs_1$$
$$Lc_3 = B_1 + Cs_1 + Er_1 + Cs_1 + Da_1$$
$$Lc_4 = Db_1 + Cs_1$$

Case D:
$$Ld_1 = T_{1.1} + Es_1 + Cs_1 + Dr_1 \quad (5.5)$$
$$Ld_2 = Ds_1 + Db_1 + Cs_1 + Er_1 + Cs_1 + Da_1$$
$$Ld_3 = Db_1 + Cs_1$$
$$Ld_4 = B_2 + Cs_1$$

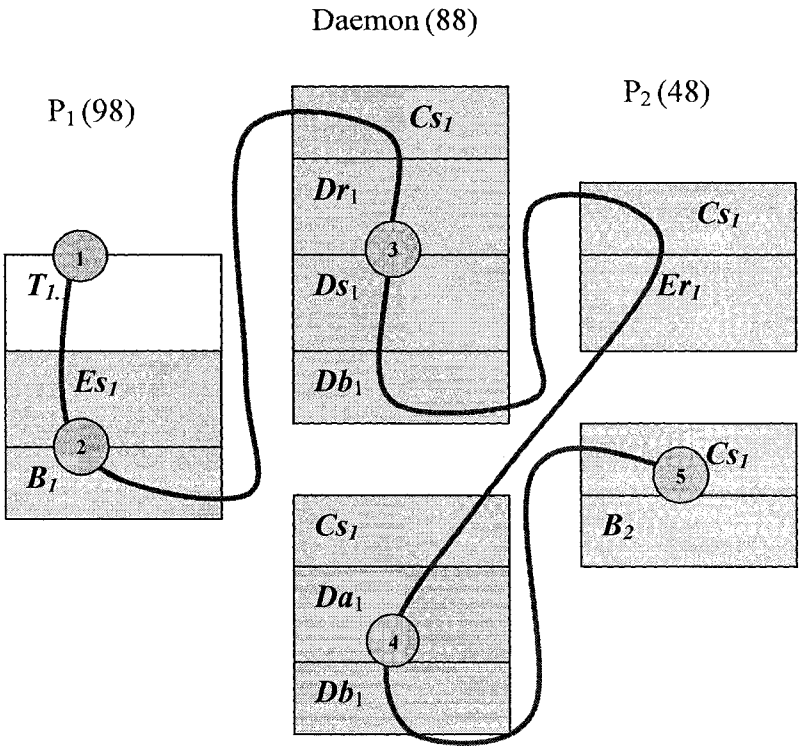

Figure 19: Processes and daemon running sequence in Case A (P_2<Daemon<P_1)

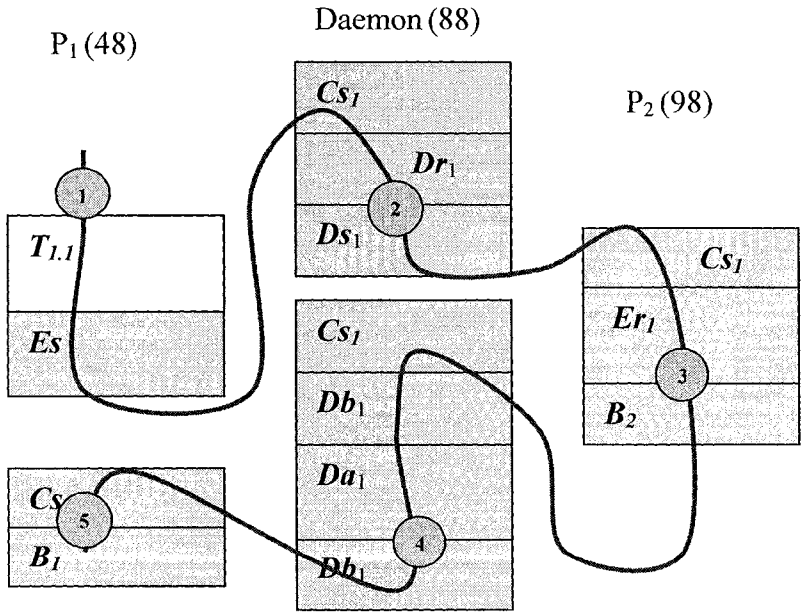

Figure 20: Processes and daemon running sequence in Case B (P_1<Daemon<P_2)

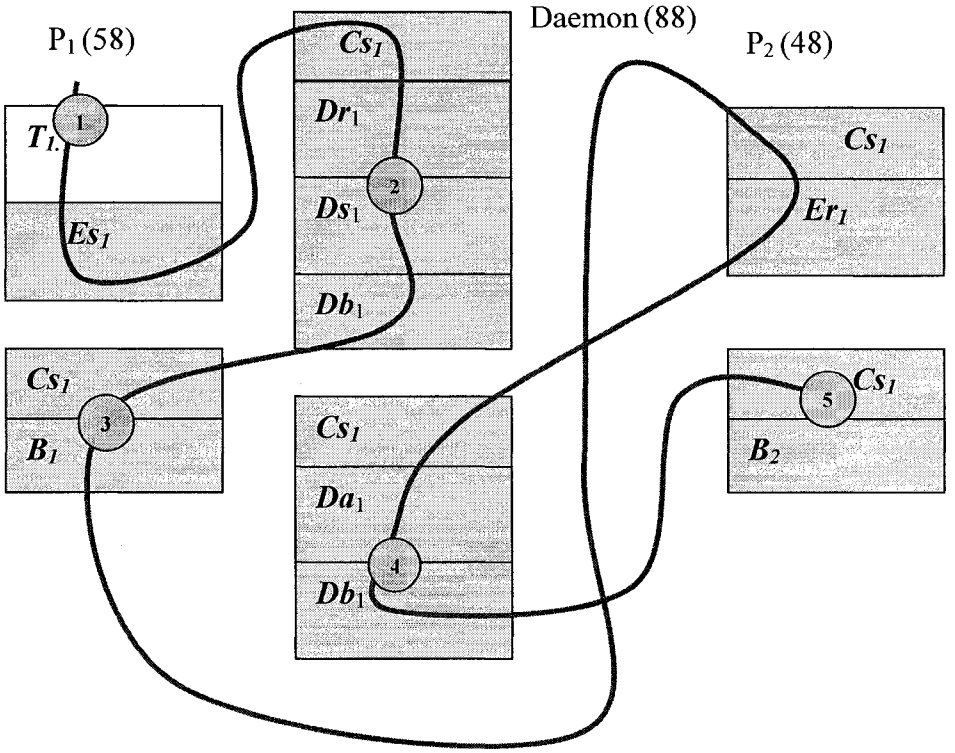

Figure 21: Processes and daemon running sequence in Case C (P_2<P_1<Daemon)

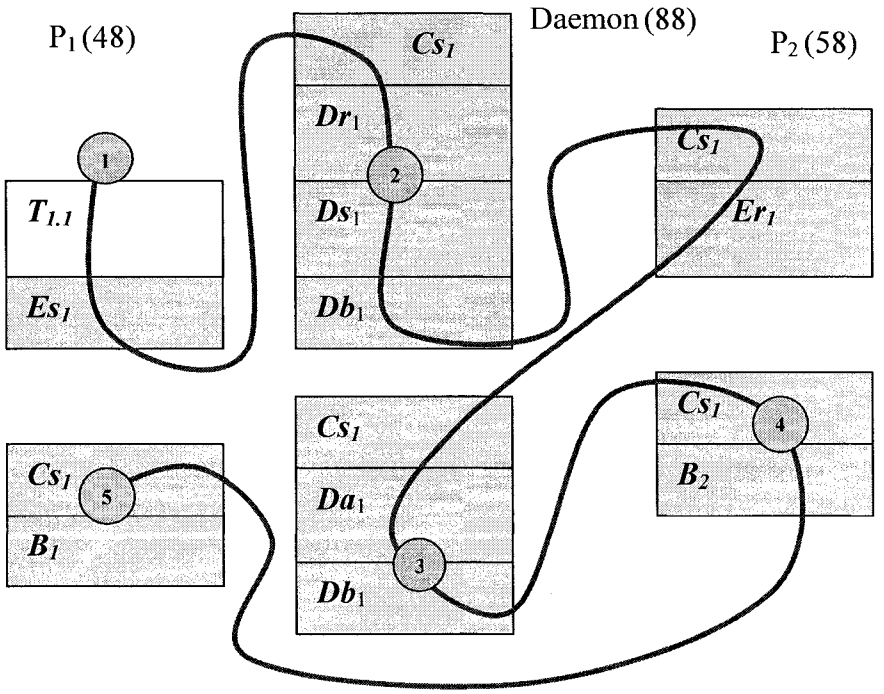

Figure 22: Processes and daemon running sequence in Case D (P_1<P_2<Daemon)

After system designers measure all latencies in Case A to D, they can use the equations 5.6, which were obtained by reducing the equations 5.2 through 5.5, to calculate all of the components of the environment overheads. The test-set for measuring message-related overheads is named as Test-Set-02.

$$Es_1 = La_1 - T_{1.1} \qquad (5.6)$$

$$Db_1 = La_4 - Cs_1$$

$$B_2 = Ld_4 - Cs_1$$

$$Da_1 = Lb_3 - Ld_4 - La_4 + Cs_1$$

$$Ds_1 = Lc_2 - La_4$$

$$Dr_1 = Lb_1 - La_1 - Cs_1$$

$$Er_1 = Lb_2 - Lc_2 + La_4 - Cs_1$$

$$B_1 = La2 - Lb_1 + La_1$$

5.3.4 Measuring Network Communication Overhead (Test-Set-03)

Network communication overhead occurs when a message is sent from one node to another through the network. The network communication overhead is written as *En*, and is the extra time cost for inter-process communication via the network as compared with the message communication within one node. The network communication overhead includes not only the time cost on the line transmission, but also the time cost of the network I/O hardware.

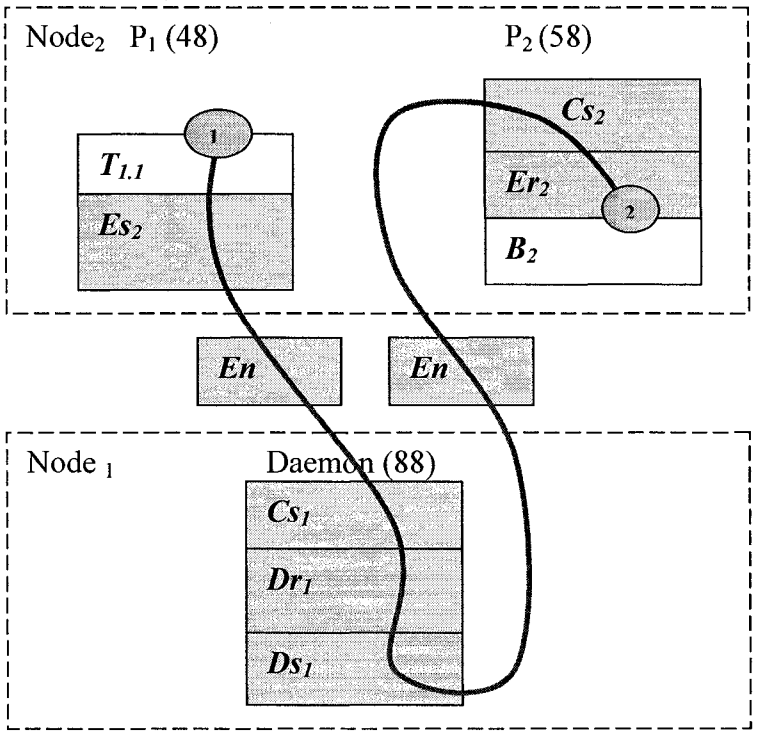

Figure 23: Measuring the network communication overhead

To measure the network communication overhead, the processes P_1 and P_2 are deployed on $Node_2$ and the middleware daemon is deployed on $Node_1$ as shown in Figure 23. The latency L_{12} from timestamp 1 to timestamp 2 is measured. Timestamp 1 is obtained before P_1 on $Node_2$ prepares to publish a message to the daemon on $Node_1$ through the network. After the daemon receives the message it forwards the message to P_2. P_2 obtains the message and sends back an ACK to the daemon. P_2 continues running to obtain timestamp 2. Note that the timing recorded by the processes on $Node_1$ does not include pre-emption by the daemon because the daemon is located on a different node. According to the process executing sequence shown in Figure 23, the latency L_{12} can be represented as equation 5.7. The network communication overhead En can be calculated

using equation 5.8, which is derived from equation 5.7. The test-set for measuring network communication overhead is named as Test-Set-03.

$$L_{12} = (T_{1.1} + Es_2) + En + (Cs_1 + Dr_1 + Ds_1) + En + (Cs_2 + Er_2) \quad (5.7)$$

$$En = (L_{12} - (T_{1.1} + Es_2) - (Cs_1 + Dr_1 + Ds_1) - (Cs_2 + Er_2)) / 2 \quad (5.8)$$

5.3.5 Hypothesis Value

Three kinds of values can be obtained while measuring temporal parameters: maximum values, minimum values and mean values. Unfortunately, all of these values have drawbacks as inputs to the prediction algorithms. Minimum values are optimistic and are therefore inappropriate for deciding whether timing constraints will be met under more strenuous conditions. Maximum values are also inappropriate to use in expressions that involve subtracting the values, since the subtraction may yield an overly optimistic value. Mean values are more stable between runs of a test, but they are too optimistic and cannot be used as an upper bound in performance analysis. Therefore, the prediction algorithms need a different kind of inputs that are both stable and can represent upper bounds on the temporal parameters.

A hypothesis value is introduced here to meet this input requirement. It combines the advantages of both mean values and maximum values, and overcomes their drawbacks. An upper ratio is the quotient of a maximum value divided by a mean value. Upper ratios are different on each test case. For example, in Test-Set-02, all 16 latencies can have different upper ratios. A hypothesis ratio is chosen to be greater than the maximum of all upper ratios in a test set. The hypothesis ratio for environment overheads

is decided by system designers. A hypothesis value is defined as a product of a mean value and a hypothesis ratio. Therefore, hypothesis values are enlarged from all of mean values in the same ratio and greater than relevant maximum values. The following equations show a summary of the definition of the hypothesis value:

Upper ratio = maximum value / mean value

Hypothesis ratio >maximum (all upper ratios in a test set)

Hypothesis value = mean value × hypothesis ratio

All the environment overheads must have the same hypothesis ratio to reduce the errors of calculations. The hypothesis ratio of processing time may be different from that of environment overhead because of their behaviour. The hypothesis values are used as pessimistic inputs to performance calculation.

5.4 Prediction Algorithms

The prediction algorithms are intended to be used by application developers to calculate application performance results. The algorithm for calculating scenario response time is described in Section 5.4.1, and considers multi-period processes. The algorithm for determining the CPU utilization is described in Section 5.4.2. The environment overheads that were introduced in previous sections are included in both algorithms. Some illustrative examples are also given to help to draw out relevant issues associated with the algorithms.

5.4.1 Scenario Response Time

The goal of this section is to provide a pessimistic algorithm for calculating the response time of a given scenario. The example scenario diagram shown in Figure 24 is provided to help in the presentation of the prediction algorithm. In the example, six processes are deployed on the three nodes to create scenarios Sc_1 and Sc_2, and the daemon is located on $Node_1$. Both of the scenarios involve process P_2, so P_2 is a multi-period process. Scenario Sc_1 contains messages M_1 and M_2, and only has one end point. The response time of scenario Sc_1 is the response time of M_2. Scenario Sc_2 contains messages M_3, M_4 and M_5, and has two end points. Since there are two end points, the scenario does not really have a single response time, and application programmers may choose to focus on either the response time of M_4 or the response time of M_5. The response times of M_4 and M_5 must be calculated separately if application developers are interested in both of them.

A multi-period process is given a different ID for each scenario in which it participates. $P_{i.n}$ represents process P_i's involvement in scenario Sc_n. For example, in Figure 24, when the multi-period process P_2 is running on the scenario Sc_1, it is called $P_{2.1}$. When P_2 is running on the scenario Sc_2, it is called $P_{2.2}$. Of course, both $P_{2.1}$ and $P_{2.2}$ are the process P_2, but $P_{2.1}$ and $P_{2.2}$ may have different process service times and period in the different scenarios.

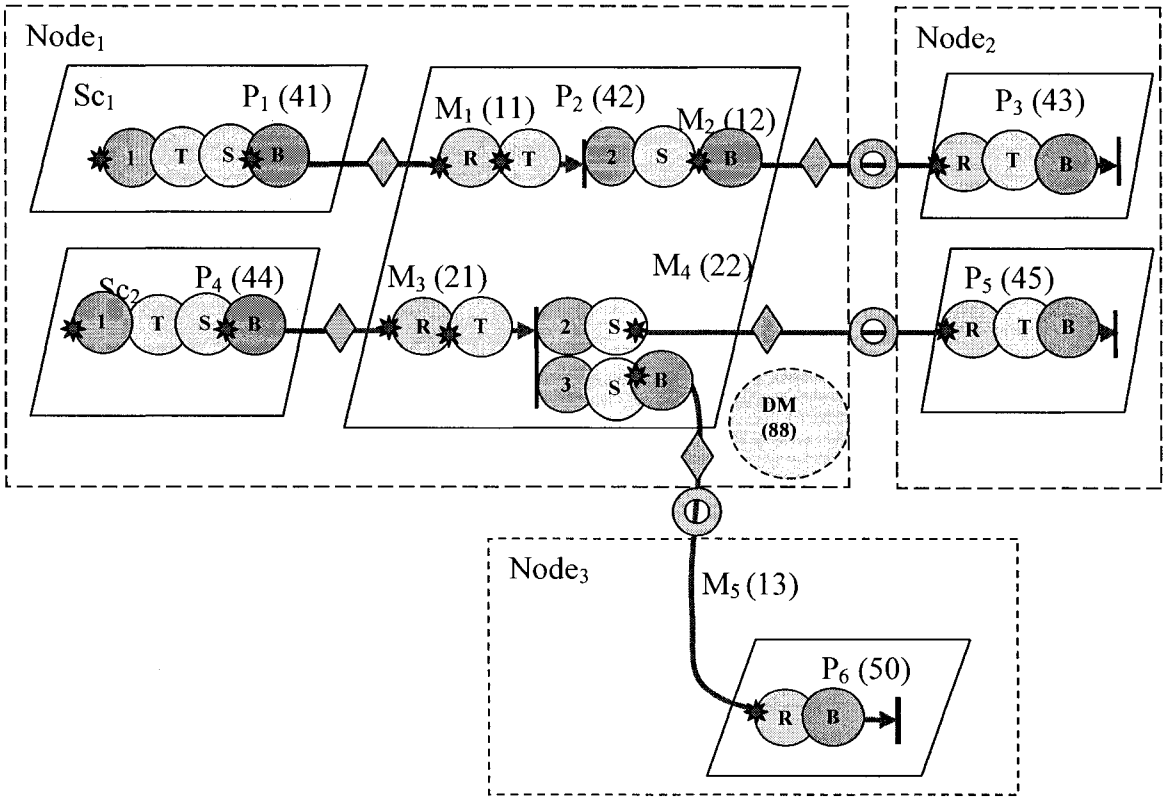

Figure 24: An example scenario diagram for performance prediction

The response time of a given scenario is the sum of four terms: the sum of delays due to the daemon, the sum of process service time of each process involved, the sum of delay caused by higher priority processes not involved directly, and the sum of network communication overheads encountered in the scenario. In general, the response time of scenario Sc_n, which is noted as Rsc_n, can be calculated by equation 5.9. The organization of the equation into the four contributing terms is described following the equation.

$$Rsc_n = \sum_{k \in mg(v)} \lceil Ps_n / Pm_k \rceil \times Dm_{v,k} + \sum_{i \in sc(n)} S_{i,n} + \sum_{j \in nd(n)} \sum_{i \in hp(n,j)} \lceil Ps_n / Ps_m \rceil \times S_{i,m} + En \times Ndistr_n \quad (5.9)$$

where:

- $mg(v)$ is the set of messages that is routed by the daemon on $Node_v$,
- Ps_n is the period of the scenario Sc_n,
- Pm_k is the publishing period of the message M_k,
- $Dm_{v,k}$ is the daemon service overhead for routing message M_k,
- $sc(n)$ is the set of processes that belong to the scenario Sc_n,
- $S_{i,n}$ is the process service time of process $P_{i,n}$ from $sc(n)$,
- $nd(n)$ is the set of nodes involved in scenario Sc_n.
- $hp(n,j)$ is the set of processes in $Node_j$ that do not belong to scenario Sc_n and whose priority is higher than or equal to the lowest priority in scenario Sc_n process on $Node_j$, if $Node_j$ does not contain any processes in the scenario then $hp(n, j)$ is empty,
- En is the network communication overhead,
- $Ndistr_n$ is the number of times the network communication overhead contributes to the response time of Sc_n.

The first term of the scenario response time in equation 5.9 is a sum of the delays due to the daemon. The ceiling function is used here to calculate the maximum number of times that the daemon service overhead can contribute to the delay. For example, in Figure 24, there are 5 messages in the diagram and all of the messages are routed by the same daemon, therefore, the message set that is routed by the daemon is $mg(v)=\{ M_1, M_2, M_3, M_4, M_5\}$. Suppose the periods of Sc_1 and Sc_2 are 100ms and 50ms respectively, and the respective publishing period of messages M_1 through M_5 are 100ms, 100ms, 50ms,

50ms and 50ms. Therefore, the respective results of calculating the ceiling function of Ps_1/Pm_k for messages M_1 through M_5 are 1, 1, 2, 2 and 2. The daemon is located on $Node_1$, and the daemon service overhead of message M_1 is noted as $Dm_{1,1}$. Because the daemon sends only one copy of each message, the daemon service overheads for all messages are the same as $Dm_{1,1}$. Therefore, the total delay caused by the daemon is: $(1+1+2+2+2)\times Dm_{1,1}$. To complete the calculation of delays due to the daemon, the delay $Dm_{1,1}$ must first be calculated.

According to the environment overhead model presented earlier, the daemon service overhead $Dm_{v,k}$ in equation 5.9 is a combination of the daemon receiving overhead, the daemon sending overhead, the daemon ACK overhead, the context switch overheads and the blocking overheads. Since the daemon is located in $Node_v$, $Dm_{v,k}$ can be calculated by equation 5.10:

$$Dm_{v,k} = (Cs_v + Dr_v) + (Ds_v \times Npub_k + Db_v) + (Cs_v + Da_v + Db_v) \times Npub_k \quad (5.10)$$

where $Npub_k$ is the number of copies of the message M_k that are published by the daemon, and the remainder of the terms were introduced in Section 5.3.1.

For example, consider the daemon service overhead cost $Dm_{1,1}$for M_1 in Figure 24. Because only one copy of message M_1 is published by the daemon, the value of $Npub_k$ is 1, and $Dm_{1,1}$ can be calculated as:

$Dm_{1,1} = (Cs_1+Dr_1)+(Ds_1+Db_1)+(Cs_1+Da_1+Db_1)$.

The second term of the scenario response time in equation 5.9 is the sum of process service times of each process in scenario Sc_n that contribute to the scenario response time. For example, consider the response time Rsc_1 of scenario Sc_1 in Figure 24. The set of processes that belong to the scenario is $sc\ (1) = \{P_{1,1}, P_{2,1}, P_{3,1}\}$. The respective

service times of the processes are $S_{1,1}$, $S_{2,1}$, and $S_{3,1}$, and the sum of the process service times in Sc_1 that contribute to the scenario response time is $S_{1,1}+S_{2,1}+S_{3,1}$.

The process service time is the sum of the processing times and overheads for that process in the given scenario. The process service time $S_{i,n}$ in equation 5.9 can be calculated by collecting all the time intervals and overheads in the scenario Sc_n within process P_i from the scenario diagram as described earlier in Section 5.3.1. For example, the process service time of $P_{1,1}$, $P_{2,1}$ *and* $P_{3,1}$ in scenario Sc_1 can be calculated as:

$$S_{1,1}= Cs_1+T_{1,1}+Es_1+ Cs_1+B_1.$$

$$S_{2,1}= Cs_1+Er_1+Cs_1+T_{2,1}+Es_1+ Cs_1+B_2.$$

$$S_{3,1}= Cs_2+Er_2+T_{3,2}+B_3.$$

Parallel execution in different nodes must be considered in calculating scenario response time. As in the previous example, the sum of individual process service times in scenario Sc_1 can simply be added together to obtain the result for the second term of equation 5.9. However, determining the response time of scenario Sc_2 is more complicated, since it has involves end-points in different processes located on different nodes. The scenario response times for the messages M_4 and M_5 must be calculated separately, and application developers need only calculate the response time for the messages they are interested in. For example, to calculate the scenario response time of M_5, the process service time of P_5, which does not contribute to the scenario response time, need not to be taken into account, because P_5 is running on $Node_2$ and is not involved in the message path for M_5. The sum of process service time in Sc_2 can be calculated as $S_{4,2}+S_{2,2}+S_{6,2}$.

The third term of the scenario response time in equation 5.9 is the delay caused by higher priority processes. For each node $Node_j$ that the given scenario involves, all of the processes not involved in the scenario but having priorities higher than or equal to the lowest priority process in scenario Sc_n on $Node_j$ can contribute to the delay. For example, in the Figure 24, the processes set $hp(1,1)$ is $\{P_{4.2}, P_{2.2}\}$, and $hp(1,2)$ is $\{P_{5.2}\}$. Suppose the periods of Sc_1 and Sc_2 are 100ms and 50ms respectively, and hence the result of the ceiling function of Ps_1/Ps_2 is 2. The total delay caused by higher priority process is $S_{4.2}\times 2+ S_{2.2}\times 2+S_{5.2}\times 2$. Therefore, in the worst case, an instance of Sc2 may be pre-empted twice by instances of Sc1, in which case the process service time $S_{4.2}$, $S_{2.2}$, and $S_{5.2}$ also can be calculated from the scenario diagram as follows:

$$S_{4.2}= Cs_1 +T_{4.3}+Es_1+ Cs_1+B_4.$$

$$S_{2.2}= Cs_1+Er_1+Cs_1+T_{2.3}+(Es_1+ Cs_1)+(Es_1+ Cs_1+B_2).$$

$$S_{5.2}= Cs_2+Er_2 +T_{5.4}+B_5.$$

The fourth term of the scenario response time in equation 5.9 is the network communication overheads encountered in the scenario. Only the network communication overheads that contribute to the scenario response time are taken into account. For example, in the Figure 24, there is only one network communication overhead in scenario Sc_1, so $Ndistr_1$ is 1 and the total network communication overhead for scenario Sc_1 is $En\times 1$.

Adding all these four terms of the scenario response time together, the total response time of scenario Sc_1 is:

$$Rsc_1= (1+1+2+2+2)\times Dm_{1.1}+(S_{1.1}+S_{2.1}+S_{3.1})+ (S_{4.2}\times 2+ S_{2.2}\times 2+S_{5.2}\times 2)+En\times 1.$$

The prediction algorithm for scenario response time relies on correctly identifying the relevant process service times and structure attributes in the scenario diagram.

5.4.2 CPU Utilization

CPU Utilization is a measure of the percentage of non-idle processing of CPU. The CPU Utilization of a given node is a sum of the contribution of each periodic process belonging to the node. For the process P_i on $Node_j$, $S_{i.n}$ represents the service time of process $P_{i.n}$ in scenario Sc_n. If the daemon is located on $Node_j$, the CPU utilization in $Node_j$ can be calculated using equation 5.11:

$$U_j = \sum_{i \in nd(j)} S_{i.n} / Ps_n + \sum_{k \in mg(j)} Dm_{j.k} / Pm_k \qquad (5.11)$$

where:

- $nd(j)$ is the set of processes located on the $Node_j$,
- Ps_n is the period of the scenario Sc_n,
- $mg(j)$ is the set of messages that is routed by daemon on $Node_j$,
- Pm_k is the publishing period of message M_k,
- $Dm_{j.k}$ is the daemon service overhead cost for a message publishing.

For example, in Figure 24, the CPU utilization of $Node_1$ is noted as U_1. The set of processes located in the $Node_1$ is $nd(1) = \{P_{1.1}, P_{2.1}, P_{4.2}, P_{2.2}\}$. The set of messages that is routed by daemon on $Node_1$ is $mg(j) = \{M_1, M_2, M_3, M_4, M_5\}$. Suppose the periods of these messages are $Pm_1 = Pm_2 = 100ms$, and $Pm_3 = Pm_4 = Pm_5 = 50ms$, the CPU utilization in $Node_1$ can be calculated according to equation 5.11 as follows:

$$U_1 = S_{1.1}/100 + S_{2.1}/100 + S_{4.2}/50 + S_{2.2}/50$$

$$+Dm_{1,1}/100+ Dm_{1,2}/100+ Dm_{1,3}/50+ Dm_{1,4}/50+ Dm_{1,5}/50$$

If the daemon is not located on $Node_j$, the daemon does not contribute utilization of CPU in the $Node_j$. The U_j can be simply calculated as equation 5.12:

$$U_j = \sum_{i \in nd(j)} S_{i,n} / Ps_n \qquad (5.12)$$

As mentioned before, PN modeling includes three parts: the visual notation, the environment overheads measurement and the prediction algorithms. All of these have been described in Sections 5.2, 5.3 and 5.4. The PN modeling procedure is summarized in the next section.

5.5 The Methodology of PN Modeling

The methodology employed for Path Network (PN) modeling is designed to meet the requirements for an ideal solution as stated in Chapter 4. There are 6 steps in the methodology shown in Figure 25. If the prediction results are feasible, in other words, if the application and the DRE environment meet the system performance requirements, only the first 5 steps are performed. Otherwise, some adjustments should be done in step 6 and then the methodology repeats from step1.

The first step of PN modeling is to build the PN model of the application. This step is performed by application developers. Application diagrams are drawn first to reflect the publish/subscribe relationships among processes and messages in different modes of the application. Then, application developers use their experience to temporarily deploy the middleware daemon and all processes on different nodes and draw a deployment diagram. If the deployment is not satisfactory, it can be adjusted in step 6

later. Finally, application developers create application diagrams on the deployment diagram to make scenario diagrams for different modes of the application. The temporal parameters, such as processing times and environment overheads, can be directly marked on each message path in the scenario diagrams. The application, deployment and scenario diagrams make up the PN model.

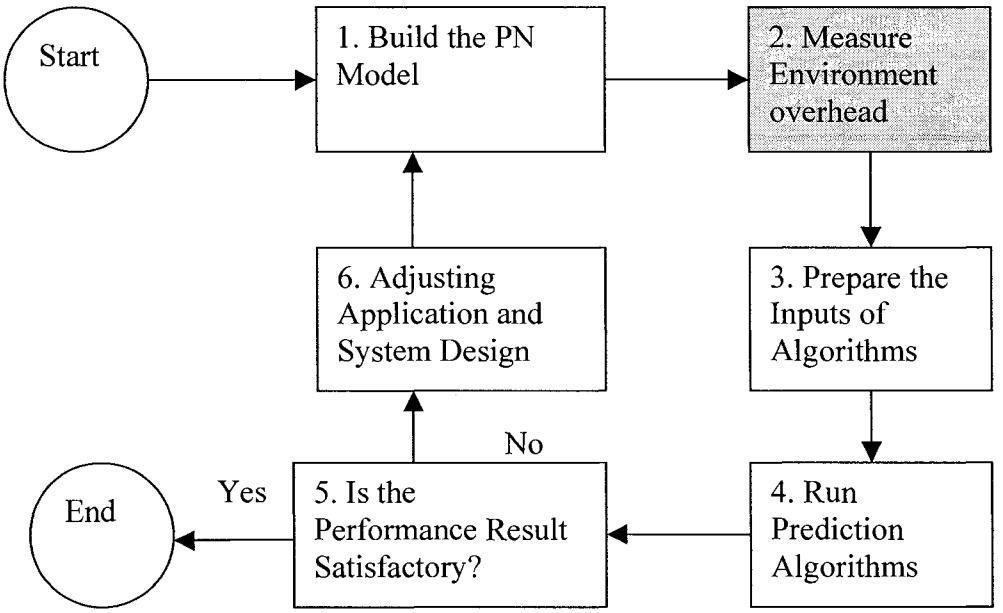

Figure 25: The procedure to use PN modeling

The second step of PN modeling is to measure the environment overheads of the target system. This is the only step performed by system designers and is marked in dark colour in Figure 25. Usually the measurement is repeated many times to obtain the values of the environment overheads. The first run of the measurement is cancelled because it is sometimes influenced by the connection set up and caches. All the measurement results are rounded to the microsecond. First, the context switch overhead is measured using

Test-Set-01. Then, the message-related overheads that occur inside the processes and the daemon are measured using Test-Set-02. Finally, the network communication overhead is measured using Test-Set-03. System designers provide the hypothesis values of each environment overheads to application developers to carry on next step.

The third step of PN modeling is to prepare the input data of the prediction algorithm. This step is performed by application developers. At the beginning, application developers measure the processing time of each process in the relevant scenarios. The hypothesis ratio of the processing time should be decided on, and the hypothesis value of each processing time should be calculated. Then, application developers should abstract all the structural attributes from the scenario diagram, such as the set of processes that belongs to the given scenario. Finally, the daemon service overheads for each message should be calculated according to equation 5.10. The process service time for each process in the relevant scenario should be calculated using the hypothesis value of environment overheads.

The fourth step of PN modeling is to run the prediction algorithm to calculate the scenario response time and CPU utilization. This step is also performed by application developers. The response time of the given scenario can be calculated according to equation 5.9, and the CPU utilization of a given node can be calculated according to equation 5.10 or 5.11 depending on the location of the middleware daemon.

The fifth step of PN modeling is to analyse the prediction results. The results are analysed by application developers. If the results of prediction satisfy the system performance requirements, the DRE environment is feasible for the application. That is the end of PN modeling for the target system. The prediction results provide application

developers with valuable information about the performance of their applications and also provides confidence in the safety of their application and the system. System designers can also use this performance information to adjust the design of the DRE architecture and make confident decisions. If the results of prediction do not satisfy the requirements, the sixth step of PN modeling must be performed.

The sixth step of PN modeling is to adjust the application or system design. The adjustments are performed by application developers. The adjustments and tuning can be done on processing time, process deployment, and or even the DRE environment. Application developers can ask system designers for upgrading the nodes with higher speed CPUs or changing to a more efficient operating system. After the necessary adjustments have been made, the PN model must be rebuilt based on the adjustments. The procedure of PN modeling is then repeated from the first step.

Chapter 6 Case Study: UAV Project

An experimental version of the GeoSurv II UAV was set up on a performance test-bed and was used as a target system to validate PN modeling. Section 6.1 describes a PN model of the UAV system. An application diagram, a deployment diagram and a scenario diagram are drawn from a subset of the UAV application design. Section 6.2 provides the values of the temporal parameters measured in the performance test-bed using the approach from Section 5.2. The calculation steps and the performance results of the UAV experimental system are given in the section 6.3. An analysis of these results is described in the final section.

6.1 The PN Model of the UAV System

According to the GeoSurv II UAV functional requirements and application design, [35] the Evasive Manoeuvring Mode is the one of the most critical situations in all six modes as described in Section 2.1. This mode is chosen as an example in the case study to show how application developers can use the visual notation of PN modeling to set up a scenario diagram. Table 3 and Table 4 [35] list the processes and messages used for obstacle avoidance according to the GeoSurv II UAV simulation functional requirements. The priority of each process and message and the publish/subscribe relationships between processes and messages are also shown in these tables.

Message ID	Message Name	Priority	Publish by	Subscribe by	Trigged by
M_1	GPS_Sensor	11	P_1	P_2	Timer
M_2	AutoGPS_Sensor	12	P_2	P_3	M_1
M_3	ObstacleDetected	13	P_3	P_4	M_2 (Found obstacle)
M_4	AutoSetAltitude (Altitude)	14	P_4	P_2	M_3&& in Critical Zone
M_5	AutoSetSpeed (Speed)	14	P_4	P_2	M_3&& in Critical Zone
M_6	AutoTurn (Heading, Bank)	14	P_4	P_2	M_3&& in Critical Zone
M_7	SetAltitude (Altitude)	17	P_2	P_5	M_5
M_8	SetSpeed (Speed)	17	P_2	P_5	M_6
M_9	Turn (Heading, Bank)	17	P_2	P_5	M_7
M_{10}	System Status (Normal, Failure)	20	P_6	P_7	System self-check Sensor

Table 3: Messages list used for obstacle avoidance [35]

Process ID	Process Name	Priority	Messages Subscribed	Messages Published
P_1	Sensor Simulator	45	N/A	M_1
P_2	Autopilot	44	M_1, M_4, M_5, M_6	M_2, M_7, M_8, M_9
P_3	Obstacle Detection	43	M_2	M_3
P_4	Autonomy	42	M_3	M_4, M_5, M_6
P_5	Actuator Simulator	49	M_7, M_8, M_9	N/A
P_6	Self-check Simulator	51	N/A	M_{10}
P_7	Emergency handling	52	M_{10}	N/A

Table 4: Processes list used for obstacle avoidance [35]

The dynamic behaviour of the Evasive Manoeuvring Mode can be described using an application diagram as shown in Figure 26. The Sensor Simulator Process P_1 is triggered by an internal timer and periodically publishes the GPS_Sensor message M_1. The Autopilot Process P_2 subscribes to M_1 and publishes the AutoGPS_Sensor message M_2. The Evasive Manoeuvring Mode begins when the Obstacle Detection Process P_3, which subscribes to M_2, finds that the UAV has just entered the critical zone. After P_3 reports the situation to the Autonomy Process P_4 by publishing the Obstacle Detection message M_3, P_4 sends command messages to the Autopilot Process P_2 to continually adjust the altitude, speed, and direction of the UAV, at the highest message publishing rate (20Hz) until the UAV leaves the critical zone and enters the safe zone. Then the UAV mode shifts back to the previous normal mode (Transit Mode or Survey Mode).

The command messages published by the Autonomy Process P_4 are the Auto Set Altitude message M_4, the Auto Set Speed message M_5 and the Auto Turn message M_6. These three messages are used to control the altitude, speed and direction of the UAV. The Autopilot Process P_2 subscribes to the three kind of command messages, and publishes the Set Altitude message M_7, the Set Speed message M_8 and the Turn message M_9. The messages M_7, M_8 and M_9 are used to control the Actuator Simulator process P_5. In the generic case, the altitude, speed and direction need not be adjusted every period, so these three commands are located in three different messages. For example, if a UAV wants to avoid a tower, it only needs to adjust speed and direction and keep the altitude stable. In the worst case, such as avoiding a mountain, all three commands should be sent every period.

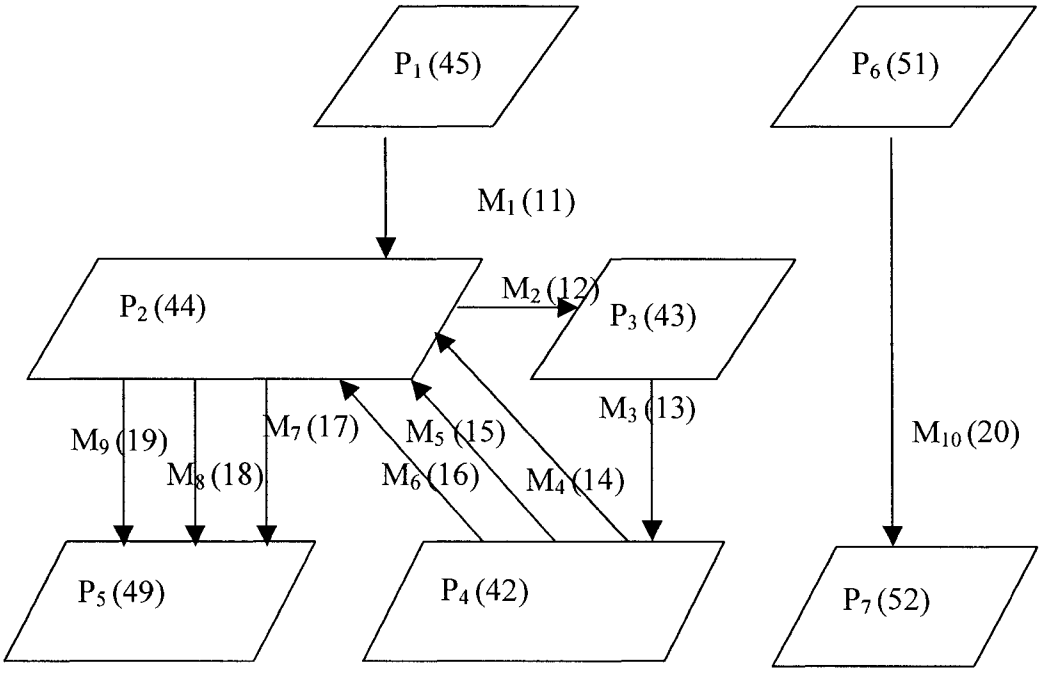

Figure 26: An application diagram for the UAV Evasive Manoeuvring Mode

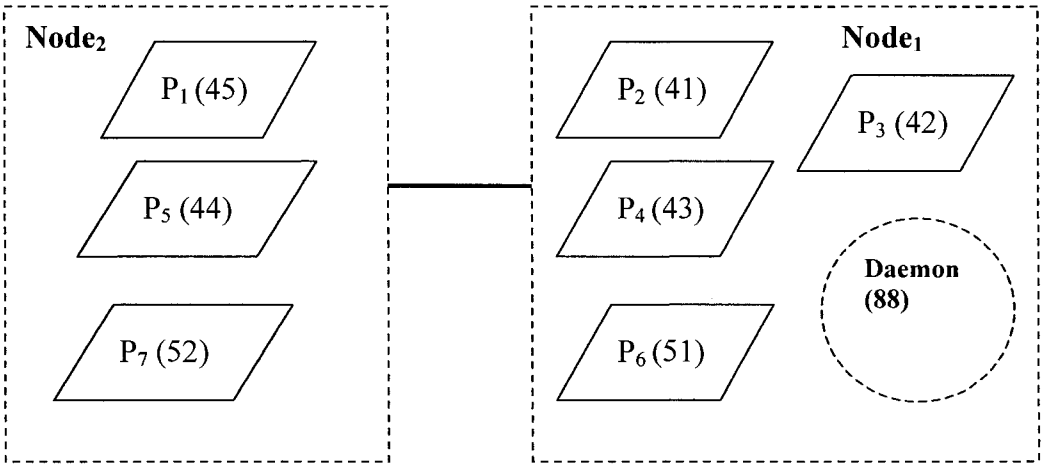

Figure 27: A deployment diagram for the UAV experimental system

A special process called the Self-check process P_6 monitors the system status, and reports the system status to the Emergency handling process P_7 periodically (at 20Hz)

using the message M_{10}. If the Emergency handling process P_7 receives a "danger" status from P_6, it controls the actuators to shut down the engine and eject the parachute for a safe landing.

From the information given by Table 3 and Table 4, it is not difficult for application developers to draw the application diagram and the deployment diagram for the UAV system. Figure 27 shows a deployment diagram for the UAV experimental system. There are only two nodes in the system. The daemon, with the highest priority (88), is deployed on the $Node_1$.

The scenario diagram for the UAV Evasive Manoeuvring Mode is shown in Figure 28. Processes P_1, P_5, P_6 and P_7 are device processes. Scenario IDs, such as Sc_1 and Sc_2, are marked near the first trigger point of each scenario. The scenario Sc_1 starts from P_1 and ends at P_5, and contains 9 message paths. Scenario Sc_2 starts from P_6 and ends at P_7, and contains only 1 message path.

The message sequence numbers are marked in the start point of each message path. The process priority and the message priority are considered to determine the message sequence numbers of the message paths, the trigger times. For example, in scenario Sc_1, there is a difficult decision need to be made regarding the sequence of message M_5 and M_7. After P_4 publishes M_4, it cannot continue publishing M_5. The priority of P_2 is higher than the priority of P_4. So the M_7 is published prior to M_5.

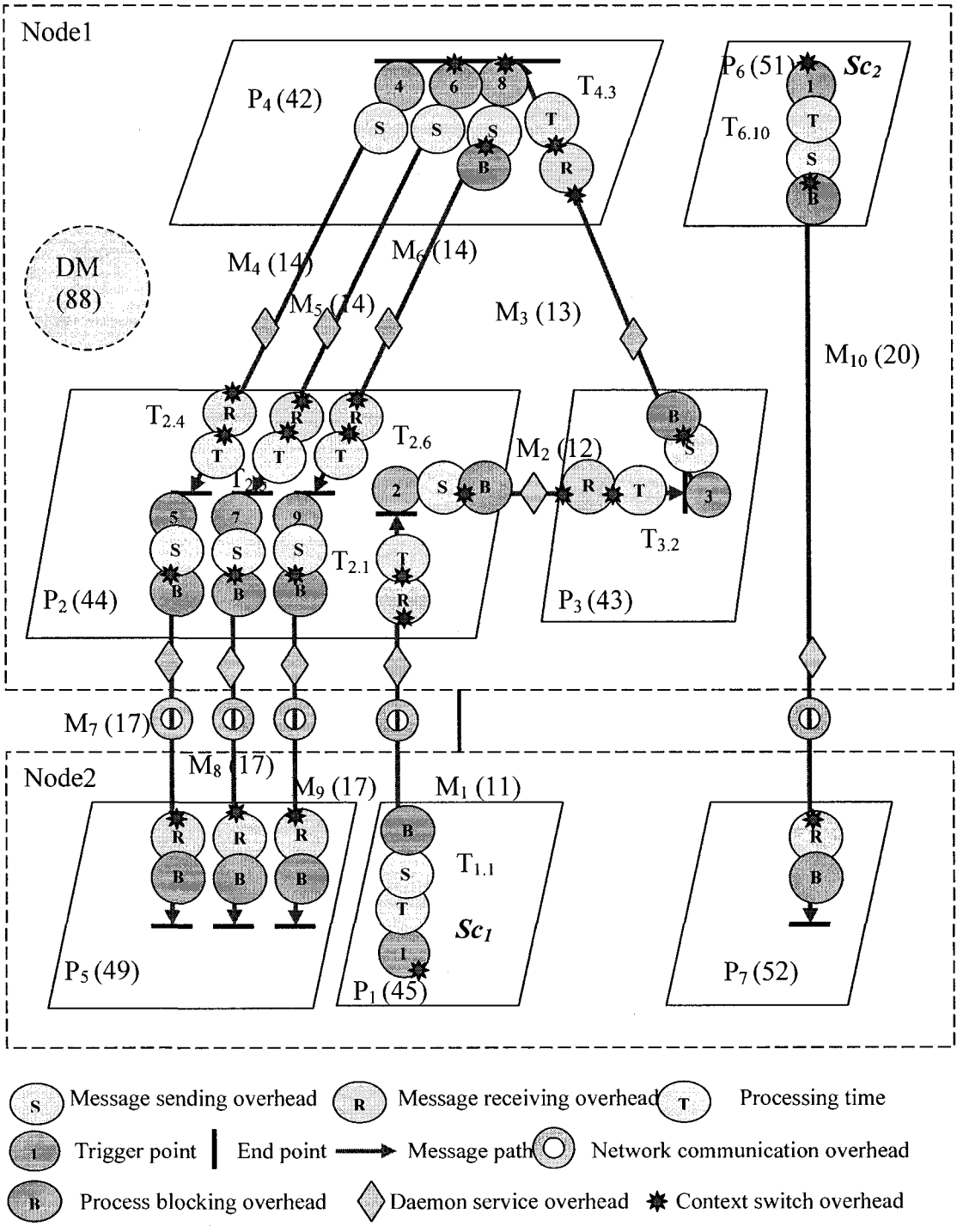

Figure 28: A scenario diagram for UAV Evasive Manoeuvring Mode

6.2 Environment Overheads of the Test-bed

Measuring the environment overheads of the test-bed is one of the most important and difficult steps for predicting the performance of the application. The test-bed consists of 2 computers interconnected using an Ethernet local area network (LAN) connected and controlled by a D-Link router DI-641+ with 10 Mbps data transfer rate. The Ethernet local area network uses the IEEE 802.3i protocol. One computer, which is called $Node_1$, has a Pentium III (Katmai) 450MHz CPU with a 512KB off-chip cache and 256MB of RAM. The system clock of $Node_1$ is 448.631MHz. The other computer, which is called $Node_2$, has a Pentium IV 1.5GHz CPU with on-die 256KB cache and 256MB of RAM. The system clock of $Node_2$ is 1483.099 MHz. The test-bed does not connect to any other computers or networks to reduce the chance of network collisions. $Node_1$ and $Node_2$ are the only two computers connected to the LAN. Irrelevant processes and applications are forbidden to run on the test-bed during the testing. Both $Node_1$ and $Node_2$ run the Red Hat Fedora 6 operating system, with the Linux 2.6 kernel. The open source middleware CMU-IPC (Version 3.7.10, Dec-30-2005) is used as the publish/subscribe middleware in the test-bed.

Linux provides a real-time scheduler based on absolute process priority [37]. Real-time priorities are in the range of 0 through 99, with higher numbers representing higher priority. If the processes have absolute priority 0, the traditional scheduling policy applies. Otherwise the real-time scheduler policy applies. When two processes are running or ready to run and both have the same absolute priority, the order of execution

on the CPU is determined by the scheduling policy. There are two available: First Come First Served (FCFS) and Round Robin (RR). The FCFS policy is used in the test-bed. Linux includes a number of daemons, which are "resident" programs that periodically check system and perform any necessary processing. They do not take any input and don't normally produce any output. The presence of Linux daemons is likely to influence some measurements adversely, but they are central to the operating system and cannot be disabled. The hypothesis value introduced in section 5.3.5 is used to counter the influence of the Linux daemon.

Intel processors provide programs with access to a time-stamp counter. The time-stamp counter keeps an accurate count of every cycle that occurs on the processor. The Intel time-stamp counter is a 64-bit MSR (model specific register) that is incremented every clock cycle. To access this counter, programmers can use the RDTSC (read time-stamp counter) instruction. This instruction loads the high-order 32 bits of the register into EDX, and the low-order 32bits into EAX. To convert the cycle counts into time units, the following equation can be used:

Number of Micro seconds (μs) = *Number of* cycles / *Main frequency (MHz)*

Case ID	Value Name	Mean Value (μs)	Hypo-thesis Value (μs)	Standard Deviation (μs)	Max Value (μs)	Upper Ratio (%)	Min Value (μs)	Lower Ratio (%)
A	La_1	54	68	1.3	58	106.9%	50	92.6%
	La_2	92	115	2.5	114	124.3%	90	97.8%
	La_3	133	166	2.0	149	112.2%	130	97.7%
	La_4	27	34	0.5	29	105.9%	26	96.3%

Table 5: Measurement data of Case A of Test-Set-02 in $Node_2$

Environmental Overhead and processing time (μs)	Hypothesis Ratio (%)	Hypothesis Value of $Node_1$ (μs)	Hypothesis Value of $Node_2$ (μs)
Cs	125%	16	9
Es	125%	106	55
Er	125%	163	75
Ds	125%	31	10
Dr	125%	120	79
Da	125%	63	35
Db	125%	49	25
B_1	125%	N/A	27
B_2, B_3, B_4	125%	30	N/A
B_6	125%	59	N/A
B_5, B_7	125%	N/A	14
En	125%	428	428
$T_{1.1}$	125%	N/A	13
$T_{6.10}$	125%	23	N/A
$T_{2.1}$, $T_{2.4}$, $T_{2.5}$, $T_{2.6}$, $T_{3.2}$, $T_{4.3}$	101%	632	N/A

Table 6: The hypothesis value of temporal parameters

In the following tests, each measurement is repeated 100 times to obtain the values of the environment overheads. The first measurement in each test is cancelled because it is sometimes influenced by the connection set up and caches. All the measurement results are rounded to the microsecond. The detail of all data measurements and calculation steps are provided in Appendix A. Table 5 lists the measurement data of Case A of Test-Set-02 on $Node_2$, and can be used to illustrate the selection of the hypothesis value. The upper ratio of La_2, which is the quotient of a maximum value divided by a mean value, is 124.3%. The lower ratio of La_2, which is the quotient of a minimum value divided by a mean value, is 97.8%. The upper ratio of La_2 is the largest upper ratio in all test sets. The hypothesis ratio for all the environment overheads is set as

125%, which is slightly larger than the maximum upper ratio of 124.3% in all test sets. The hypothesis value is the produce of the mean value and 125%. For example, the hypothesis value of La_2 is 115, which is also a little larger than the maximum value of La_2 114.

The confidence in measurements can be obtained from the statistical confidence intervals for the measured results. For example, the mean value of La_2 in Table 5 is 92. The lower and upper 99% confidence interval (CI) for La_2 is 91.15 and 92.45. This means that there is 99% certainty that the true mean is somewhere between 91.15 and 92.45. This result is good because it give a high probability that the hypothesis value is being calculated based on a realistic representative of the true mean.

Further confidence in the selection of the hypothesis value can be obtained from the standard deviation of the measured results. Chebyshev's inequality entails the bounds for all distributions for which the standard deviation is defined [38]. It indicates that at least 98% of the values are within 7 standard deviations from the mean. For example, in Table 5, the mean value of La_2 is 92 and the standard deviation of La_2 is 2.5. Therefore, the upper and lower bound of 7 standard deviations is 109.5 and 74.5. This means that 98% of the measured values of La_2 will fall between 74.5 and 109.5. This shows that the hypothesis value of La_2, which is 115, is larger than the 7 standard deviation upper bound of 109.5. This adds further confidence that the hypothesis value is pessimistic, since it is likely to be larger than 98% of measured values.

The hypothesis ratio for $T_{1.1}$, and $T_{6.10}$ is also set as 125%, and hypothesis ratio for all other processing times is set at 101% because of the low difference between the maximum and minimum value. Table 6 is a summary of hypothesis values of all the

temporal parameters measured from the test-bed according to the methodology described in Chapter 5.

6.3 Performance of the UAV System

After the PN model and the value of all parameters are available, the next step is to use the PN model, temporal parameters of the test bed, and algorithms in Section 5.4 to calculate and predict the UAV performance.

6.3.1 UAV Scenario Response Time

The response time of scenario Sc_1 is predicted using the prediction algorithms in Section 5.4 and the scenario diagram of the UAV application shown in Figure 28. Both scenarios Sc_1 and Sc_2 are running on 20Hz, so the message-publishing periods in both scenarios are 50ms. P_1 and P_6 are released at the same time and lead to a worst-case situation happening as desired.

The first term of the scenario response time from equation 5.9 is a sum of the delays happening on the daemon. The message set that is routed by daemon is $mg(1)=\{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, M_{10}\}$. The periods of Sc_1, Sc_2 are 50ms. The results of ceiling function of Ps_1/Pm_k for messages M_1 through M_{10} are all 1. The daemon is located on $Node_1$, and the daemon service overhead of the message M_1 is written as $Dm_{1,1}$. Because the daemon sends one copy of each message, the value of $Npub_k$ is 1 and the daemon service overheads for all messages are same as $Dm_{1,1}$. The total delay caused

by the daemon can be calculated as: $10 \times Dm_{1.1}$. $Dm_{1.1}$ can be calculated using equation 5.10 as follows:

$$Dm_{1.1} = (Cs_1 + Dr_1) + (Ds_1 + Db_1) + (Cs_1 + Da_1 + Db_1)$$

$$= 16 + 120 + 31 + 49 + 16 + 63 + 49 = 344\ \mu s$$

The second term of the scenario response time from equation 5.9 is a sum of the process service times of each process in scenario Sc_1. The set of processes that belongs to scenario Sc_1 is $sc\ (1) = \{P_{1.1}, P_{2.1}, P_{3.1}, P_{4.1}, P_{5.1}\}$. The process service time of process $P_{1.1}$, $P_{2.1}$, $P_{3.1}$ $P_{4.1}$, *and* $P_{5.1}$ are $S_{1.1}$, $S_{2.1}$, $S_{3.1}$, $S_{4.1}$, and $S_{5.1}$. The sum of the process service times in Sc_1 is $S_{1.1}+S_{2.1}+S_{3.1}+S_{4.1}+S_{5.1}$. These process service times can be calculated by collecting all the time interval and overheads in the scenario Sc_1 within each process from the scenario diagram as follows:

$$S_{1.1} = T_{1.1} + Es_2 + B_1 = 13 + 55 + 27 = 95\ \mu s;$$

$$S_{2.1} = ((Cs_1 + Er_{1+}\ Cs_1) + T_{2.1} + (Es_{1+}\ Cs_1 + B_2))$$

$$+ ((Cs_1 + Er_{1+}\ Cs_1) + T_{2.4} + (Es_{1+}\ Cs_1 + B_2))$$

$$+ ((Cs_1 + Er_{1+}\ Cs_1) + T_{2.5} + (Es_{1+}\ Cs_1 + B_2))$$

$$+ ((Cs_1 + Er_{1+}\ Cs_1) + T_{2.6} + (Es_{1+}\ Cs_1 + B_2))$$

$$= ((16 + 163 + 16) + 632 + (106 + 16 + 30)) \times 4$$

$$= (195 + 632 + 152) \times 4 = 997 \times 4 = 3916\ us;$$

$$S_{3.1} = (Cs_1 + Er_{1+} Cs_1) + T_{3.2} + (Es_{1+}\ Cs_1 + B_3)$$

$$= (16 + 163 + 16) + 632 + (106 + 16 + 30) = 997\ \mu s;$$

$$S_{4.1} = (Cs_1 + Er_{1+} Cs_1) + T_{4.3} + Es_1 + (Cs_1 + Es_1) + (Cs_1 + Es_1 + Cs_1 + B_4)$$

$$= (195 + 632 + 106 + 122 + 168) = 1223\ \mu s;$$

$$S_{5.1} = Cs_2 + Er_2 = 9 + 75 = 84\ \mu s;$$

When the process service time $S_{5.1}$ is calculated, the parallel execution of the two nodes must be considered. Process P_5 is triggered by M_7, M_8 and M_9 separately in scenario Sc_1. The process service time of P_5 triggered by M_9 is the only one that needs to be taken into account, because the service time of P_5 triggered by M_7 and M_8 is executed parallelly with the service time of P_2, P_4 and the daemon, and the later is much larger than the former. The maximum time of two parallel-executed service times should always be chosen and the other should be omitted.

The third term of the scenario response time from equation 5.9 is the delay caused by higher priority processes. The processes set *hp(1,1)* is $\{P_{6.2}\}$, and the processes set *hp(1,2)* is $\{P_{7.2}\}$. Because the result of the ceiling function of Ps_1/Ps_2 is 1, the total delay caused by higher priority process is $S_{6.2}\times 1+ S_{7.2}\times 1$. The $S_{6.2}$ and $S_{7.2}$ also can be calculated from the scenario diagram as following:

$S_{6.2}= Cs_1 +T_{6.10}+Es_1+ Cs_1+B_6=16+23+106+59=204\ \mu s;$

$S_{7.2}= Cs_2+Er_2+B_7=9+75+17=101\ \mu s.$

The fourth term of the scenario response time from equation 5.9 is the sum of network communication overheads encountered in the scenario. Only two network communication overheads, which are triggered by M_1 and M_9, could influence the response time of scenario Sc_1. The network communication overheads triggered by M_7 and M_8 are running in parallel with the process service times and are omitted. The $Ndistr_1$ is 2, and sum of the network communication overhead for scenario Sc_1 is $En\times 2=428\times 2=856\ \mu s$.

Adding all these four part of the scenario response time together, the response time of the scenario Sc_1 is:

$Rsc_1 = 10 \times Dm_{1.1} + (S_{1.1} + S_{2.1} + S_{3.1} + S_{4.1} + S_{5.1}) + (S_{6.2} + S_{7.2}) + En \times 2$

$= (344 \times 10) + (95 + 3916 + 997 + 1223 + 84) + (204 + 101) + 428 \times 2$

$= 3440 + 6315 + 305 + 856 = 10916\ \mu s$

So the predicted UAV scenario response time of Sc_1 is 10916 μs.

6.3.2 UAV CPU Utilization

Predicting the CPU utilization in $Node_1$ is a little easier when compared to the scenario response time. The CPU utilization in $Node_1$ is written as U_1. The set of processes located on $Node_1$ is $nd(1) = \{P_{2.1}, P_{3.1}, P_{4.1}, P_{6.2}\}$. The set of messages that are routed by the daemon on $Node_1$ is $mg(1) = \{M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8, M_9, M_{10}\}$. The periods of these messages and processes are all *50ms(or 50000μs),* and the CPU utilization in $Node_1$ can be calculated according to equation 5.11 as follows:

$U_1 = S_{2.1}/50000 + S_{3.1}/50000 + S_{4.1}/50000 + S_{6.2}/50000 + (Dm_{1.1}/50000) \times 10$

$= (3916 + 997 + 1223 + 204 + 3440)/50000 = 9780/50000 = 19.6\%.$

So the predicted UAV CPU utilization of $Node_1$ is 19.6%.

6.4 Analysis of the Case Study Results

PN modeling includes three parts: the visual notation, the environment overheads measurement, and the prediction algorithms. This section provides a full analysis of all these three parts as applied to the case study of the GeoSurv II UAV project.

6.4.1 Analysis of the Visual Notation

The visual notation of PN modeling is used to build a PN model of the UAV system includes generating application, deployment and scenario diagrams. The first two diagrams are not difficult for application developers. The scenario diagram is more complicated because it needs to show all the elements and their relationships including symbols that represent the temporal parameters. All the structure attributes come from the scenario diagram.

The scenario diagram provides a simple and easy way for application developers to view the PN model of the target system and to abstract the structure attributes of the application. With the scenario diagram, application developers can easily know the relationship among processes, messages, nodes and the daemon. The process priorities and message priorities are marked on the diagram to help application developers determine the message sequence easily. The different kinds of environment overheads and the process processing times are also shown as easy-identified icons on the diagram. The dependencies among messages are clear shown and application developers can easily determine the preceding and succeeding messages of the current message according to the scenarios. A scenario usually starts from a sensor and end at actuators, and application developers can calculate the process service time for a scenario by adding the processing times and overheads on the trace of the scenario within the process.

Because all the steps of building the PN model are done manually, it is simple for application developers to set up the PN model for a small-scale DRE system that contains only several processes, messages and nodes. However, if the target system is a large-scale

system, the efficiency is lower. Thus, the generalization of environment overheads icons is desirable. The compression and refining of these icons can be done in the future research. At the same time, further research into making the modeling automatic is also desirable.

6.4.2 Analysis of the Environment Overhead Measurement

The environment overhead is broken down into several pieces for system designers to measure easily. Ideally, a worst-case response time of a scenario could be given if the worst-case values of each temporal parameter were known. Unfortunately, it is impossible in practice to obtain the exact worst-case values through limited measurement times. The maximum value obtained from the case study is actually a maximal observed value. Using the maximum value is still not safe for hard real-time systems, [36] because the maximum value could be different for the same latency in different groups of measurements. The value of temporal parameters based on the maximum latency is not stable in the different measurements. The final performance results that come from this maximum value only provide to the application developers with a general idea about the response time of the application.

The hypothesis value is introduced to obtain a pessimistic estimate performance result and give more confidence to application developers. The hypothesis value relates to the mean value and the hypothesis ratio. The mean values are much more stable in different groups of measurements as compared to the maximum values. The hypothesis ratio of the processing times can be different from that of the overheads. The hypothesis

ratio should be set a bit larger than the maximum of all upper ratios. There is no accurate standard formula for how to decide the hypothesis ratio. This decision usually relies on the experience and confidence of application developers and system designers. If the hypothesis ratio is too large, the performance results will be overestimated.

The test-bed in the case study is not an ideal DRE environment because of constraints of time, cost and equipment of the experiment. Future research should use a more deterministic test-bed that combines with a token ring LAN protocol (IEEE 802.5) network, vxWorks or QNX Real-time Operating System (RTOS), and commercial Data Distribution Service (DDS) publish/subscribe middleware. Such a test-bed could obtain more accurate temporal parameter measurement results and lead to a more accurate prediction result.

6.4.3 Analysis of the Prediction Algorithms

The procedure of calculating the performance results in the case study is simple. It does not need special software tools and expensive instruments, does not need deep knowledge of the operating system and hardware, and does not need deep knowledge of mathematics and statistics. All the step of the calculation can be performed manually, and it is easy for application developers to learn. Given the structure attributes of processes, messages and nodes, and the measurement value of the temporal parameters of the target system, it is possible for application developers to calculate the system performance using the prediction algorithms and the scenario diagram. However, collecting the service time of each process that contribute to the scenario response time from the scenario diagram

still requires that application developers understand the sequence of parallel execution on different nodes. As the example in Section 6.3 shows, when the process service time $S_{5.1}$ is calculated, some process service time in process P_5 will not be taken into account because of the parallel execution of the two nodes. Future work should be done for develop an algorithm that can omit the service time of parallel executing processes.

The pessimistic performance prediction is much more useful than the optimistic prediction for a hard real-time system like the UAV, because the user requires the validation that the system always meets the time constraints. The pessimistic prediction results are larger than the performance measurement results. If the pessimistic prediction results still meet the performance requirement, the developers gain more confidence on the application software.

The pessimistic prediction also can reduce the complexity of the prediction algorithms. For example, in equation 5.9 for predicting the scenario response time, the daemon service overheads for all messages in the system are taken into account, even though some messages may not be published within the scenario response time, and the daemon service overheads for these messages do not contribute to the scenario response time. This makes the prediction results a little larger than the measurement results. However, application developers can benefit from the simple algorithm and need not to find out which messages do not influence the scenario response time. Another example also shows the simplification. In equation 5.9 for predicting the scenario response time, the ceiling function determines the number of times that the daemon or higher processes contribute the scenario response time. The number of times is the worst-case number and

larger than the actual number. However, application developers can use simple addition to calculate the delay instead of iterative approach.

Performance metrics	Maximum measurement result (Mr)	Hypothesis prediction result (Hp)	Difference (Hp-Mr)/Mr (%)
Scenario response time	8499us	10916us	*28.4%*
CPU Utilization	15.9%,	19.6%	*23.3%*

Table 7: Summary of the case study results

The summary of the case study result can be found in Table 7. The hypothesis prediction results are larger than the maximum measurement results, and also provide a pessimistic system performance. The difference for the scenario response time is 28.4%, and the difference for the CPU utilization is 23.3%. The hypothesis value based performance results provide a general guideline, rather than a strict upper bound, to help application developers gain more confidence in the performance of the UAV application. The prediction is made according to a rough and ready practical rule, not based on formal methods and exact measurements. The prediction result is not intended to be strictly accurate or reliable for every situation. Even though there is no guarantee that the hypothesis result will be an absolutely safe upper bound on the performance, the hypothesis prediction result can still help application developers gain confidence in the safety of the UAV system.

Chapter 7 Conclusions and Future Directions

Path Network (PN) modeling is an analytic approach for predicting whether a loosely coupled DRE environment such as that of the GeoSurv II UAV will meet its performance goals. Determining the performance of the GeoSurv II application is a challenging aspect of development. Path Network (PN) modeling meets this challenge. The visual PN notation emphasizes the aspects of the application and its DRE environment that contribute to application performance. Algorithms are presented for measuring the temporal properties of the environment and calculating the scenario response times and CPU utilization. The algorithms are designed to separate the concerns of application developers from those of system designers. This separation allows application developers, who may have little knowledge of real-time systems, to gain valuable information about the performance of the GeoSurv II UAV application. Furthermore, this performance information will be essential input for the design of the on-board DRE architecture.

As a result of this research, three major contributions have been made in the visual notation, the environment overhead measurement and the prediction algorithms. The visual notation of PN modeling is designed to capture the relevant aspects of the GeoSurv II UAV application, its network architecture and deployment that are needed for GeoSurv II application performance analysis. The PN model identifies the relevant aspects of the environment overhead encompassing the CMU-IPC middleware, the

operating system and network communications, and gives algorithms for measuring each aspect of the environment overheads. The prediction algorithms is designed for calculating the scenario response times and CPU utilization of an application represented as a PN model.

As mentioned before, the test-bed of the case study is not an ideal loosely coupled DRE environment because of constraints of the time, cost and equipments. Even though the test-bed is not an ideal loosely coupled DRE environment, the results of case study still indicate that PN modeling provides a simple and easy way for application developers to predict the performance of the on-board loosely coupled DRE system for the GeoSurv II UAV project.

As initial research, PN modeling is the first step in predicting the performance of the GeoSurv II UAV on-board DRE system. The generalization of environment overheads icons is desirable. The compression and refining of these icons can be done in the future research. Further research into making the modeling automatic is desirable. This includes developing a tool that can generate the scenario diagram, collect service time of each process while considering parallel execution and calculate the prediction result automatically. Future research should use a more deterministic test-bed to obtain more accurate temporal parameter measurement results that would lead to a more accurate prediction result. Future research for a more generic modeling approach could also be done. Some limitation in the thesis may be relaxed, such as the mono-core or mono-CPU restrictions, the fixed publishing period, and the fixed size of messages.

References

[1] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating System Concepts*, Sixth Edition, John Wiley & Sons, Inc. 2002.

[2] Jane W.S. Liu, *Real-Time Systems,* Prentice-Hall, Inc. 2000.

[3] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications,* Kluwer Academic Publishers, 1997.

[4] Phillip A. Laplante, *Real-Time System Design and Analysis,* Third Edition, John Wiley & Sons, Inc. 2004.

[5] Gerardo Pardo-Castellote, "OMG Data-distribution Service (DDS): Architectural Overview". Real-Time Innovations, Inc. 2007.

[6] Mohamed Anis Mastouri and Salem Hasnaoui, "Performance of a Publish/Subscribe Middleware for the Real-time Distributed Control System", *IJCSNS International Journal of Computer Science and Network Security*, VOL.7 No.1, Jan. 2007. pp. 313-318

[7] Patrick Th. Eugster , Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, " The Many Faces of Publish/Subscribe", *ACM computing Surveys*, Vol 35, No. 2 June 2003.

[8] Reid Simmons, "Inter Process Communication (IPC) User Manual ", Carnegie Mellon University, 2005. [Online]. Available: http://www.cs.cmu.edu/afs/cs/project/TCA/www/ipc/ipc.html, [Accessed Aug 8, 2008].

[9] Object Management Group, "Data Distribution Service for Real-Time System," Version 1.2, Jan. 2007. [Online]. Available: http://www.omg.org/. [Accessed Aug. 8, 2008].

[10] Object Management Group, "Common Object Request Broker Architecture: Core Specification" Version 3.0.3, March 2004. [Online]. Available: http://www.omg.org/. [Accessed Aug. 8, 2008].

[11] Microsoft Corporation, "DCOM Technical Overview", November 1996. [Online]. Available: http://msdn2.microsoft.com/en-us/library/ms809340.aspx. [Accessed Aug. 8, 2008].

[12] Hossam A. Gabbar, *Modern formal methods and applications*, Published by Springer Publishing Company, 2006.

[13] Daniel A. Menascé; Virgilio A.F. Almeida; Lawrence W. Dowdy, *Performance by Design: Computer Capacity Planning by Example,* Pearson Education, Inc. 2004.

[14] Grady Booch, James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide,* Second Edition, Pearson Education, Inc, 2005

[15] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik, *Quantitative System Performance Computer System Analysis Using Queueing Network Models,* Prentice-Hall, Inc., in 1984

[16] C. L. Liu, and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Envirnment" *Journal of the Association for Computing Machinery,* Vol. 20, No. 1, January 1973. pp. 46-61.

[17] Murray Woodside, and Greg Franks, "Tutorial Introduction to Layered Modeling of Software Performance", Carleton University, October 23, 2007.

[18] Ti-Yen Yen, and Wayne Wolf, "Performance Estimation for Real-time Distributed Embedded System", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 11, November 1998. pp. 1125-1136.

[19] Vikram S. Adve, and Mary K. Vernon "Parallel Program Performance Prediction Using Deterministic Task Graph Analysis", *ACM Transactions on Computer Systems,* Vol. 22, No. 1, February 2004. pp. 94-136

[20] Angelo Corsaro, "Quality of Service in Publish/Subscribe Middleware," PrismTech Inc., April 26, 2006.

[21] Sun Microsystems, Inc. Java Message Service (JMS) , [Online]. Available: http://java.sun.com/products/jms/. [Accessed Aug. 8, 2008].

[22] Daniel I. Katcher, Hiroshi Arakawa, and Jay K. Strosnider, "Engineering and Analysis of Fixed Priority Schedulers", *IEEE Transactions On Software Engineering,* Vol. 19, No. 9, September 1993. pp. 920-934.

[23] Concepcio Roig, Ana Ripoll, and Fernando Guirado, " A New Task Graph Model for Mapping Message Passing Applications," *IEEE Transactions On Parallel And Distributed Systems,* Vol. 18, No. 12, December 2007. pp. 1740-1753.

[24] Luca Zanolin, Carlo Ghezzi, Luciano Baresi, "An Approach to Model and Validate Publish/Subscribe Architectures," *Specification and Verification of Component-Based Systems (SAVCBS) Workshop,* 2003. pp. 35-41.

[25] Ragunathan Rajkumar, Mike Gagliardi and Lui Sha, "The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation", *Proceedings of the Real-Time Technology and Applications Symposium*, 1995. pp. 66.

[26] M. Harkema et al. "Middleware Performance: A Quantitative Modeling Approach," *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS),* 2004. pp.733-742.

[27] Tom Verdickt et al., "Modelling the performance of CORBA using Layered Queuing Networks", *Proceedings of the 29th EUROMICRO Conference: New Waves in System Architecture,* 2003. pp. 117.

[28] David B. Stewart, "Measuring Execution Time and Real-Time Performance", *Embedded Systems Conference,* Boston, September 2006.

[29] Kevin Lai and Mary Baker, "A Performance Comparison of UNIX Operating Systems on the Pentium", *Proceedings of the USENIX 1996 Annual Technical Conference,* San Diego, California, January 1996. pp. 22.

[30] Larry McVoy, "lmbench: Portable Tools for Performance Analysis", *Proceedings of the USENIX 1996 Annual Technical Conference,* San Diego, California, January 1996

[31] D. Amyot et al., "Use Case Maps for the Capture and Validation of Distributed Systems Requirements", *Proceedings of the 4th IEEE International Symposium on Requirements Engineering,* 1999.

[32] Ugo Buy, Robert Sloan, "A Petri-net-based approach to real-time program analysis", *IWSSD '93: Proceedings of the 7th international workshop on Software specification and design,* December 1993. pp. 56-60.

[33] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour," *Proceedings of 10th Real-Time Systems Symposium,* December 1989. pp. 166-171

[34] Carleton University UAV Project, [Online]. Available: http://uav.mae.carleton.ca/.[Accessed Aug. 8, 2008].

[35] Sean Bassett, Private Communication, May 2008.

[36] Andreas Ermedahl, and Jakob Engblom, "Execution time Analysis for Embedded real-time systems", in *Handbook of Real-time and embedded systems*, Chapman and Hall/CRC, 2008.

[37] GNU project, The GNU C Library, [Online]. Available: http://www.gnu.org/software/libc/manual/html_mono/libc.html [Accessed Aug. 8, 2008].

[38] Wikipedia, Standard deviation, [Online]. Available: http://en.wikipedia.org/wiki/Standard_deviation#Relationship_between_standard_deviation_and_mean. [Accessed Aug. 8, 2008].

Appendix A Case Study Measurement Data

The measurement data and some sample calculation steps for the temporal parameters in the case study in Chapter 6 are provided. All the calculation results for the temporal parameters are summarized in Table 5 in Chapter 6.

Table A.1 lists the measurement data of each processing time in the PN model. The hypothesis ratio for $T_{1.1}$, and $T_{6.10}$ is set as 125%, and the hypothesis ratio for other processing times is set as 101% because of the low upper ratio. These measurements are preformed by application developers.

Processing Time	Mean Value (μs)	Hypo-thesis Value (μs)	Standard Deviation (μs)	Max Value (μs)	Upper Ratio (%)	Min Value (μs)	Lower Ratio (%)
$T_{1.1}$ *in* $Node_2$	10.4	13	0.5	11.5	110.2%	8.7	83.7%
$T_{6.10}$ *in*$Node_1$	18.6	23	1.53	21.9	117.7%	16.7	89.8%
$T_{2.1}$, $T_{2.4}$, $T_{2.5}$, $T_{2.6}$, $T_{3.2}$, $T_{4.3}$	625.9	632	0.15	626.2	100.05 %	625.6	99.9%

Table A.1: Measurement data of processing time

Table A.2 lists the measurement data of Test-Set-01 related to context switch overheads. The maximum upper ratio is 111.4%, and hypothesis ratio can be set to 125%. According to equation 5.1, the hypothesis value of context switch overhead in $Node_1$ is: Cs_1= *(66-11-24)/2=16μs,* and the hypothesis value of the context switch overhead in $Node_2$ is: Cs_2= *(38-7-14)/2=9μs.*

Node Name	Over-head Name	Mean value (μs)	Hypo-thesis Value (μs)	Standard Deviation (μs)	Max Value (μs)	Upper Ratio (%)	Min Value (μs)	Lower Ratio (%)
$Node_1$	Pipe Read	8.7	11	0.33	9.2	105.6%	6.9	79.3%
	Pipe write	19.2	24	0.65	21.1	109.8%	17.7	92.2%
	Start to end	53.0	66	1.08	55.7	105.1%	49.6	93.6%
$Node_2$	Pipe Read	5.2	7	0.22	5.8	111.4%	4.6	88.5%
	Pipe write	11.1	14	0.49	12.0	108.6%	9.8	88.3%
	Start to end	30.2	38	0.59	31.8	105.4%	28.3	93.7%

Table A.2: Measurement data of Test-Set-01 related with context switch overhead

Case ID	Value Name	Mean Value (μs)	Hypo-thesis Value (μs)	Standard Deviation (μs)	Max Value (μs)	Upper Ratio (%)	Min Value (μs)	Lower Ratio (%)
A	La_1	103	129	3.0	110	106.8%	96	93.2%
	La_2	156	195	2.6	161	103.2%	151	96.8%
	La_3	272	340	4.6	306	112.4%	267	98.2%
	La_4	52	65	0.4	53	102%	51	98%
B	Lb_1	212	265	3.5	221	104.5%	203	95.8%
	Lb_2	166	208	3.5	190	114%	160	96.4%
	Lb_3	126	158	0.9	131	104.3%	124	98.4%
	Lb_4	50	63	0.3	51	101.5%	49	98%
C	Lc_1	212	265	5.8	229	108.1%	198	93.4%
	Lc_2	75	94	1.1	78	103.6%	73	97.3%
	Lc_3	234	293	5.0	268	114.4%	227	97%
	Lc_4	53	66	0.5	54	102.0%	51	96.2%
D	Ld_1	209	261	5.8	220	105.4%	196	93.8%
	Ld_2	271	339	5.1	310	114.3%	265	97.8%
	Ld_3	51	64	0.3	52	101.5%	50	98%
	Ld_4	37	46	0.5	38	103.0%	36	97.3%

TableA.3: Measurement data of Test-Set-02 in $Node_1$

Table A.3 lists the measurement data of Test-Set-02, which relates the environment overhead in $Node_1$. The maximum upper ratio is 114.4%. The hypothesis ratio is set as 125%. The following calculation steps for environment overheads use the hypothesis values as inputs to equation 5.6. Hypothesis value of $T_{1,1}$ in $Node_1$ is 23 μs.

$Es_1 = La_1 - T_{1,1} = 129 - 23 = 106\ \mu s$

$Db_1 = La_4 - Cs_1 = 65 - 16 = 49\ \mu s$

$B_2 = Ld_4 - Cs_1 = 46 - 16 = 30\ \mu s$

$Da_1 = Lb_3 - Ld_4 - La_4 + Cs_1 = 158 - 46 - 65 + 16 = 63\ \mu s$

$Ds_1 = Lc_2 - La_4 = 96 - 65 = 31\ \mu s$

$Dr_1 = Lb_1 - La_1 - Cs_1 = 265 - 129 - 16 = 120\ \mu s$

$Er_1 = Lb_2 - Lc_2 + La_4 - Cs_1 = 208 - 94 + 65 - 16 = 163\ \mu s$

$B_1 = La_2 - Lb_1 + La_1 = 195 - 265 + 129 = 59\ \mu s$

Case ID	Value Name	Mean Value (μs)	Hypo-thesis Value (μs)	Standard Deviation (μs)	Max Value (μs)	Upper Ratio (%)	Min Value (μs)	Lower Ratio (%)
A	La_1	54	68	1.3	58	106.9%	50	92.6%
	La_2	92	115	2.5	114	124.3%	90	97.8%
	La_3	133	166	2.0	149	112.2%	130	97.7%
	La_4	27	34	0.5	29	105.9%	26	96.3%
B	Lb_1	122	156	1.3	127	103.8%	119	97.5%
	Lb_2	75	94	1.9	92	121.7%	73	97.3%
	Lb_3	66	83	1.0	68	103.4%	63	95.5%
	Lb_4	28	35	0.5	30	104.3%	27	96.4%
C	Lc_1	122	156	1.4	126	103.0%	118	96.7%
	Lc_2	35	44	0.6	36	104.0%	34	97.1%
	Lc_3	119	149	3.0	141	118.4%	116	97.5%
	Lc_4	26	33	0.4	27	104.0%	25	96.2%
D	Ld_1	124	155	1.3	126	12.3%	119	96.0%
	Ld_2	132	165	2.1	149	113.1%	129	97.7%
	Ld_3	27	34	0.4	28	104.4%	26	96.3%
	Ld_4	18	23	0.4	20	105.7%	17	94.4%

Table A.4: Measurement data of Test-Set-02 in $Node_2$

Table A.4 lists the measurement data of Test-Set-02 in $Node_2$. The maximum upper ratio is 124.3%. The hypothesis ratio is set as 125% here. The following calculation steps for the environment overheads use the hypothesis value as input to equation 5.6. The hypothesis value of $T_{1.1}$ in $Node_2$ is 13 μs.

$Es_2 = La_1 - T_{1.1} = 68-13 = 55\ \mu s$

$Db_2 = La_4 - Cs_2 = 34-9 = 25\ \mu s$

$B_2 = Ld_4 - Cs_2 = 23-9 = 14\ \mu s$

$Da_2 = Lb_3 - Ld_4 - La_4 + Cs_2 = 83-23-34+9 = 35\ \mu s$

$Ds_2 = Lc_2 - La_4 = 44-34 = 10\ \mu s$

$Dr_2 = Lb_1 - La_1 - Cs_2 = 156-68-9 = 79\ \mu s$

$Er_2 = Lb_2 - Lc_2 + La_4 - Cs_1 = 94-44+34-9 = 75\ \mu s$

$B_1 = La_2 - Lb_1 + La_1 = 115-156+68 = 27\ \mu s$

Value Name	Mean Value (μs)	Hypo-thesis Value (μs)	Standard Deviation (μs)	Max Value (μs)	Upper Ratio (%)	Min Value (μs)	Lower Ratio (%)
L_{12}	940.0	1175	96.1	1151.2	122.5%	679.2	72.3%

Table A.5: Measurement data of Test-Set-03 for the network latency

Table A.5 lists the measurement data of Test-Set-03 for the network latency. The hypothesis ratio for the network communication overhead can also be set as 125%. According to equation 5.8, the hypothesis value of the network communication overhead En can be measured and calculated as follows:

$En = (L_{12} - (T_{1.1} + Es_2) - (Cs_1 + Dr_1 + Ds_1) - (Cs_2 + Er_2))/2$

$= (1175 - (13+55) - (16+120+31) - (9+75))/2$

$= (1175-68-167-84)/2 = 428\ \mu s$