a

a

a

a

Acknowledgments

I thank my advisor, Dr. Arun Lakhotia, for his valuable guidance, extraordinary support, inspiration, and encouragement. He was always there to support during my highs and lows in the research period. This thesis would never have been conceptualized without his motivation and the ideas he provided me. My gratitude for him cannot be expressed in a paragraph.

Special thanks to Pablo Mejia for the expert guidance he provided me throughout the thesis. I want to thank Suresh Golconda, Amit Puntambekar, and Santhosh Padmanabhan for reviewing my thesis document and providing constructive comments. Also, I thank other members of Team CajunBot for their valuable feedback.

Special thanks to Santhosh Padmanabhan, for providing constant support and encouragement right from my initial days in the country and for sharing every moment of my joys and disappointments.

I would like to express my thanks to all my friends at Lafayette for all those memorable times and fun trips.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

The development of Autonomous Ground Vehicles (AGV) faces a growing complexity with components distributed across multiple machines, running on heterogeneous environments connected using various network topologies with widely differing bandwidths. The computational power required for processing sensor data, fusing the data from various sensors, and making appropriate navigation decisions in real-time is huge, thereby justifying the need for distributing the computations across multiple machines.

The need arises for an intermediate agent to facilitate communication among the different components in the distributed environment so they function as an integrated system. The term "middleware" refers to a software system that acts as an intermediary between different application components in a distributed system. The application components, which may be running on different operating systems, need to communicate with each other seamlessly, with minimal communication overhead.

The middleware should provide support for various issues that need to be addressed while developing an autonomous system. For example, in case of an AGV, when two or more sensors are producing sensor data, it is necessary to fuse data from multiple sensors for the data to be used by other processes. When two sources generate data at different frequencies, it may not always be appropriate to use the most recent data from both the sources. Doing so may lead to the fusion of mutually inconsistent data. Along the same lines, instead of using the data generated directly by a source, sometimes it is preferred to interpolate the data for the specific time when data from another source is produced. These capabilities prove to be crucial in terms of improving the specific abilities of the autonomous vehicle. The middleware should provide interpolation and consistent data fusion support for these kinds of issues.

In a distributed environment, especially when the system is very large, monitoring the status of processes on different machines and tracking the messages generated by each process becomes a tedious and error-prone task. When an autonomous vehicle is in action, real-time monitoring of the vehicle becomes extremely important and useful to analyze the behavior of the vehicle. Therefore, the middleware should provide the necessary functionalities to deliver real-time debugging and monitoring capabilities.

While developing the software system of an autonomous vehicle, logging data generated by processes running on the vehicle during autonomous runs is essential for post-analysis of the data. Post-analysis of the data helps in identifying the behavior of the vehicle during an autonomous run. One of the most important components of the middleware is a centralized server that is dedicated to logging data, which will be used typically for post-analysis.

Thus, a middleware must take into account all of these issues and provide the necessary capabilities to function as an integrated system.

1.2 Research Objectives

The aim of this research is to propose and implement a distributed middleware that provides a transparent communication infrastructure for processes running on multiple machines to exchange information. The middleware should provide centralized logging of data and real-time monitoring and debugging capabilities in a complex distributed environment. It should satisfy the necessary Quality-Of-Service (QoS) requirements in terms of minimal communication latency and minimal packet loss.

1.3 Research Contributions

The ability for processes running on single or multiple computers to exchange information seamlessly in a complex distributed environment is critical for the successful operation of autonomous robots. The middleware module CBWare provides an efficient and transparent communication infrastructure for information exchange among multiple machines so that the producers and consumers of data are independent of each other. It supports fusion of data from multiple sensors with varying frequencies and latencies based on time of production of the data, thereby ensuring fusion of mutually consistent data. CBWare facilitates remote real-time monitoring of the system by periodically transmitting the data generated by the processes over a wireless network.

The thesis deals with the following aspects of the CBWare.

- Exchange information on a single machine.
- Exchange information across multiple machines.
- Exchange information among processes on heterogeneous platforms
- Centralized logging of data for post-analysis
- Centralized debugging capability in a distributed environment.
- Real-time remote monitoring of an autonomous system.

1.4 Significance of the Research

The proposed middleware was used in CajunBot, a six-wheeled AGV developed by the University of Louisiana at Lafayette for the Defense Advanced Research Projects Agency (DARPA) Grand Challenge 2005 [1]. As the complexity of the system increased, for example, when additional sensors were added, a single machine could not handle the voluminous data from various sources. Hence, processing needed to be done in parallel, distributing the computations on multiple machines. CBWare enabled easy scalability of computational power by seamlessly distributing the application components on multiple computers. The sensor (and other data) fusion support provided by CBWare has been an important contributor in CajunBot's ability to leverage rough terrain to increase its visibility. The remote real-time monitoring capability provided by CBWare was extremely useful in debugging and the real-time visualization of data helped in tuning various algorithms and parameters. The log data from the central log server was very useful in post-processing and post-analysis of the behavior of the vehicle. CBWare also facilitated communication among processes on heterogeneous systems dealing with byte-order differences and endian issues.

1.5 Organization of Thesis

Chapter 2 provides a brief description of the various communication paradigms used for information exchange among processes. It also discusses the existing middlewares for real-time distributed systems. Chapter 3 describes the physical architecture of the distributed system of CajunBot and the distribution of processes on various machines in CajunBot. Chapter 4 introduces the CBWare architecture, describes CBQueues, an interface used for inter-process communication on a single machine, discusses CBPackets, an interface used for exchanging information among multiple machines, and describes the data encoding format used to convert the data into a machine-independent neutral format so that the data can be transferred across the network. It also discusses the logging of data to the central log server and the remote real-time monitoring capability of CBWare. Chapter 5 evaluates the middleware on various parameters like transmission delay across machines, the order of packets received on each machine, the bandwidth of the network, and the packet rate.

Chapter 6 gives the conclusions and summarizes some limitations of CBWare and potential future work to address these limitations. The appendix provides translations routines for a message type used in the CajunBot software system.

2 Background and Related Work

Several communication paradigms have been proposed for information exchange among processes. This section discusses the various communication frameworks that have been used for Interprocess Communication. It also discusses several Inter Process Communication toolkits that have been developed over the years, some of which have been developed specifically for robotics.

The commonly used models for distributed interaction are:

- Shared Memory Model;
- Publish/Subscribe Model;
- Client/Server Model;
- Message Queues;
- Remote Procedure Calls; and
- Unix Interprocess Communication Mechanisms.

CBWare works on the Publish/Subscribe Model and uses a combination of shared memory and network communication for inter process communication in a distributed environment. This chapter elaborates on the various models listed above with a detailed discussion of the Shared Memory Model in Chapter 4.

2.1 Publish/Subscribe Model

In this model as shown in Figure 1, the producers publish data and consumers subscribe to data they require through a neutral intermediary, known as the “middleware,” so that the

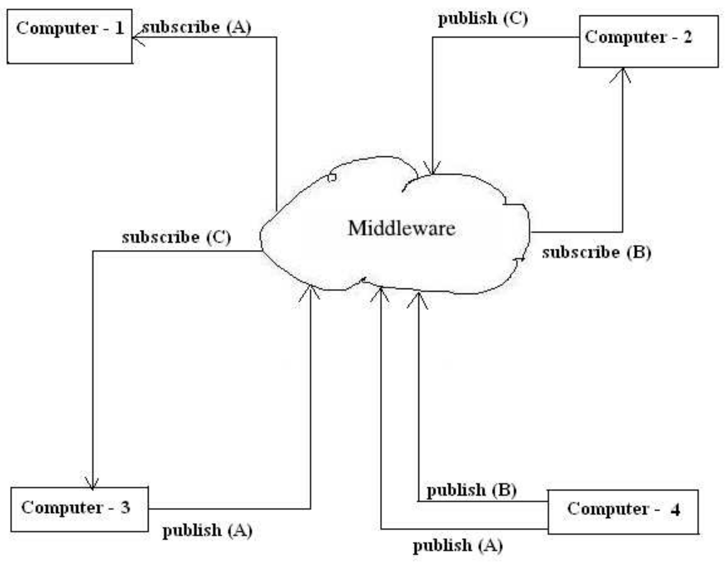

Figure 1: The Publish/Subscribe Model

publishers and subscribers of information are anonymous to each other. The main features of the Publish/Subscribe Model and some of the research issues have been discussed in [2].

The producers do not know the destination of the data and the consumers are not aware of the source of the data, thereby achieving decoupling between the publishers and the subscribers. This decoupling feature is very effective in systems with many-to-many interactions and increases the scalability of distributed systems. A detailed discussion of the other communication paradigms and the various dimensions of decoupling can be found in [3].

2.2 Client/Server Model

In this model, a module (client) sends a request to another module that acts as a server. The server processes the request from the client and returns the response back to the client. A server can receive requests from multiple clients. However, this type of information exchange is not suitable for robotic applications that are time-critical because the client blocks until it receives a response from the server and hence cannot do any useful processing in the time it waits for a response from the server. The Client/Server Model is also commonly known as the Request/Response Model.

2.3 Message Queues

Message Queues are a mechanism by which two or more processes exchange information through a common system message queue. Each message in the queue is given an identification so that processes can read the required messages. A variable number of messages, each message of a variable length, can be stored in the Message Queues. The messages stored in the queues are in First In, First Out (FIFO) order. Some of the commercial implementations of the Message Queues are IBM's WebSphere MQ and Microsoft's Message Queues.

2.4 Remote Procedure Calls

Remote Procedure Call (RPC) is a protocol through which a process running on one machine makes a call to a procedure on another machine connected through a network. When a procedure is invoked on another machine, the parameters of the procedure are passed from the caller environment to the environment where the procedure is to execute. After execution, the results are passed back to the caller environment. RPC can also be used by processes that share the same address space to exchange information. In this scenario, the procedure call will refer to a procedure that is local to the machine. RPC typically uses the Client/Server model described in Section 2.2 for information exchange. RPC is a popularly used mechanism in distributed computing. A detailed description about the implementation of Remote Procedure Calls can be found in [4].

2.5 UNIX Interprocess Communication Mechanisms

Unix offers several mechanisms for Interprocess Communication (IPC). Some of the popular IPC mechanisms are as follows.

- **Signals:** Signals are a mechanism employed by a process to raise a signal and deliver it to another process. The signal may be of any kind, for example, to interrupt another process from its current execution. The process that receives the signal has a signal handler routine that handles the received signal and acts appropriately.

- **Pipes:** Pipes are unnamed files used as I/O channels between two processes, one of which writes to the pipe and the other reads from the pipe.

- **File Locking:** In this mechanism, a file is shared among many processes that write the data that needs to be shared to the file. Write access to the file is synchronized through locks that can be set on the files.

- **Sockets:** Sockets are mechanisms that are usually used for communication between two processes running on two different machines over the network. A socket acts as a channel through which information exchange takes place between processes.

2.6 Review of Inter Process Communication Toolkits

This section reviews several Inter Process Communication toolkits developed over the recent years.

2.6.1 Object Oriented Toolkit for Inter Process Communication

The Object Oriented Toolkit for Inter Process Communication (IPT) [5] was developed in 1996 for use in an Unmanned Ground Vehicle [6]. It is an object-oriented, message-passing toolkit designed specifically for robotics. IPT has a centralized server process that performs two basic functions:

1. Initiates point-to-point connection between modules, and
2. Establishes a consistent mapping between message names and unique message ID's.

CBWare also provides the feature of establishing a mapping between message names and message numbers to ensure consistency across all modules in the distributed system. Apart from the two functions mentioned above, the IPT server also acts a log repository used to track down problems in the system. IPT uses TCP/IP Sockets for inter-machine message passing and Unix Domain Sockets for intra-machine communication. However, when two modules exist on the same machine, message passing through Unix Domain Sockets is not the most efficient mechanism to exchange information. In this mechanism, each message must be sent to every module, even though the same message could reach each module by updating a single shared memory area. IPT does not provide fault-tolerance in cases when the IPT server, which is the key element in establishing connections, might fail.

2.6.2 Inter Process Communication

The Inter Process Communication library (CMU-IPC) [7] works on the Publish/Subscribe model for communication between distributed heterogeneous processes in a large networked system. It also supports the Client/Server paradigm to exchange information. CMU-IPC is supported on multiple machine types and various operating systems. It has a centralized server, which performs the function of message passing between modules. The server also logs message traffic and other system-wide information.

2.6.3 Real-Time Communications

Real-Time Communications (RTC) [8] is a protocol developed to provide fast and reliable data delivery in real-time applications. RTC uses the Shared Memory model to deliver high bandwidth data, for example data from the sensors, from one process to another. It uses the TCP protocol to distribute data between processes on different machines. RTC is not suitable for time-critical applications where getting most of the data in a timely fashion is more important than getting all of the data in order. Since RTC uses the TCP protocol for data transfer, there may be situations where a process could be blocked from doing critical computations due to the time spent in re-transmitting lost packets. In cases where a process sends status signals that are not critical to a remote monitoring station (typically through a wireless network), TCP adds a lot of communication overhead. In such scenarios, UDP is a better choice for non-critical data transfer since it does not care about retransmission of lost packets when the wireless network fails. Another problem with RTC is that the packing and unpacking of the data transferred across the network needs to be done by the processes that send and receive information. If this conversion of data were done by the RTC protocol, the processes would spend more time in doing critical computations.

2.6.4 Network Data Distribution System

The Network Data Distribution System (NDDS) [9] works on the Publish/Subscribe model where the producers and consumers of data are independent of each other. Each processor runs an NDDS agent that acts a broker for various types of information. The consumers of a particular information type register with the broker, which sets up direct UDP connection with the producer(s) of that information type. Although NDDS was primarily designed for communication among modules connected by a network, it is not suitable for applications where processes on a single machine want to communicate with each other. In such scenarios, using shared memory for data transfer between processes on a single machine is a better option. CBWare uses shared memory for interprocess communication on a single machine and UDP protocol to transfer data across the network.

2.6.5 The Neutral Message Language

The Neutral Messaging Language (NML) [10] uses a combination of shared memory and UDP based network communication to exchange information in a distributed environment. Run-time allocation of processes to processors is enabled dynamically by means of a configuration file that contains the protocol specifications and mapping between processes and processors. This approach has two problems. When the number of processors increases, maintaining the configuration file by the user becomes a tedious and error-prone task. Also, once a message is read, it is deleted from the shared-memory buffer, making it unavailable for other processes that may want to read the message at a later point of time.

2.6.6 MIRO - Middleware for Mobile Robots

MIRO [11] is a distributed object-oriented framework developed specifically for use in mobile robots based on CORBA technology [12]. MIRO logs all the data to disk in the form of files for post-analysis and has the capability to replay the logged data from files in a timely manner. The receiver of the data cannot distinguish between logged data and data generated in real-time. This transparent nature of MIRO helps in visualizing live data as well as replaying data from logged files for post-analysis. CBWare shares this transparency feature with MIRO and additionally has the capability to sample data used for visualization at specific intervals.

2.6.7 Broker - An Interprocess Communication for Multi-Robot Systems

Broker [13] is a framework developed for communication among distributed processes in robotic systems. There is a centralized server called the “broker” that is responsible for transferring data in the form of packets between processes. All the communication takes place in the form of UDP packets sent from the publishers to the server and in turn is sent to all the subscribers of the data by the server. The server also logs all communications for later analysis. The Broker framework suffers from a major drawback, namely that the failure of the broker brings down the whole system since the entire system is dependent on the broker and there are no point-to-point connections involved. Another problem with the broker network is that routing of all the messages to the proper destinations involves two hops, from the publisher to the server and from the server to the subscribers. This becomes a serious bottleneck in terms of network congestion when the number of processes increases. Broker does not provide any support for marshaling of network data.

A detailed discussion on the various Inter Process Communication toolkits can be found in [14]. It also discusses toolkits based on other communication paradigms and gives a qualitative comparison of the various paradigms. Matteucci [15] provides a compared analysis of the various Publish/Subscribe middlewares that have been used in robotics.

3 Software Architecture and Design Issues in Distributed Systems

This chapter introduces the need for distributed system in autonomous vehicles. It explains the onboard computing system and the onboard software architecture of CajunBot. Finally, it discusses the various design issues involved in dealing with a distributed system.

3.1 Software Architecture of an Autonomous Ground Vehicle

The software system of an AGV consists of various computationally intensive processes, each of which processes huge amounts of data generated by the onboard sensors and other hardware required to drive the vehicle. The major software components of an AGV are:

- Obstacle Detector;
- Path Planner;
- Steering Controller;
- Drivers;
- Data Logger; and
- Middleware.

Figure 2 graphically depicts the software architecture of an AGV. The Obstacle Detector uses the sensor data to detect obstacles and to identify unnavigable regions of the terrain. The Path Planner gives a path around the obstacles. The Steering Controller generates commands to drive the vehicle in the path given by the Path Planner. The modules labeled Obstacle Detection and Path Planner perform the core computations necessary to drive the

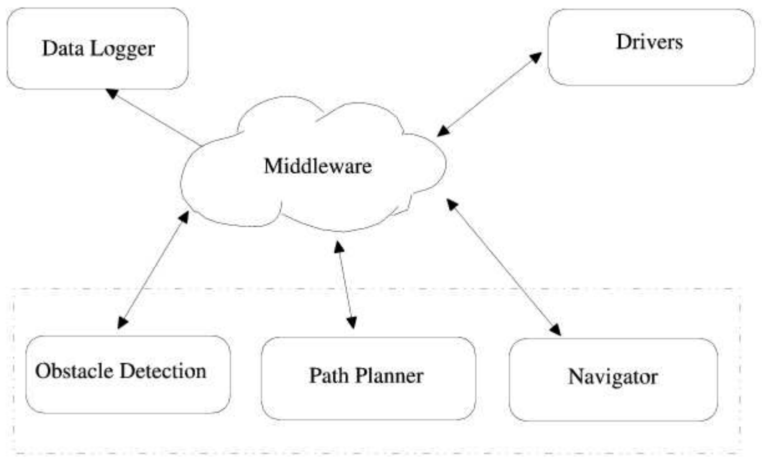

Figure 2: Software Architecture of an AGV

vehicle in autonomous mode. Apart from these two components, there are other drivers in the system that communicate with the hardware devices and convert the data to match the conventions used in the rest of the system. The Data Logger logs data generated by these modules to the disk. The Middleware provides the infrastructure for communication among these distributed modules and allows the various processes to exchange information at ease.

Each of these modules requires a fair amount of CPU time to perform the computations efficiently. It is difficult to achieve fairness when all these processes run on the same machine. Adding more sensors to achieve greater granularity of terrain and obstacle data increases the overall performance of the system, but adds to the computational complexity. Hence, there is a definite need to distribute the processes across multiple machines and meet the Quality-of-Service (QoS) requirements. The next section describes the distributed system architecture of CajunBot, with the distribution of processes on multiple machines.

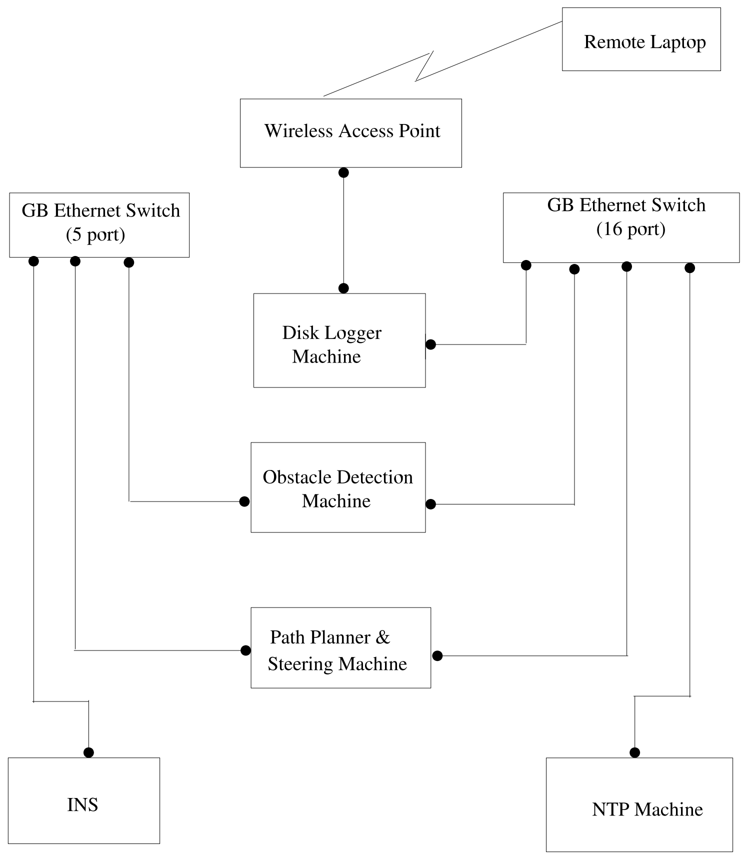
Remote Laptop
Wireless Access Point
GB Ethernet Switch (5 port)
GB Ethernet Switch (16 port)
Disk Logger Machine
Obstacle Detection Machine
Path Planner & Steering Machine
INS
NTP Machine

Figure 3: Onboard Computing System of CajunBot

3.2 Onboard Computing System of CajunBot

Figure 3 shows the computing system of CajunBot, which is a collection of four machines. These machines together provide the necessary computational power required for the software system of CajunBot. The computers labeled “Obstacle Detection Machine” and “Path Planner and Steering Machine” are Dell Poweredge 750s. The other two machines, “Disk Logger Machine” and “NTP Machine,” are Mini-ITX boards.

The sensors are connected directly to the “Obstacle Detection Machine,” which runs the Obstacle Detection module. This module is responsible for processing the data from the various sensors and computing the terrain and obstacle information. The obstacle information data are used by the Local Planner and Navigator modules that run on the “Path Planner and Steering Machine” to compute a path around the obstacles and drive the vehicle in the planned path. The “Disk Logger Machine” is used for logging data generated from various processes during a run, typically for post-analysis. The “NTP machine” provides Network Time Protocol service, a service necessary to synchronize data from multiple sensors and computers. The system has three networks. The first network connects the Dell Poweredge computers and the INS. This network is a 5-port Gigabit Ethernet Switch. Since the data generated from the INS is used by all phases of the processing, its timely availability on all machines is crucial, justifying a separate network. The second network is through the 16-port Gigabit Ethernet Switch, which connects all the computers together. The third network, through the wireless access point, consists of only the disk logger machine on board. The disk logger transmits data over this wireless network for real-time remote monitoring of the system, typically from a laptop.

3.3 Design Issues in Distributed Systems

Some of the important design issues to address when dealing with a distributed system are listed below.

1. System Shutdown: Shutting down the entire distributed system with graceful termination of all processes.

2. Synchronization: Synchronizing programs and configuration files across all machines.

3. Monitoring Process Status: Monitoring the status of each process running on different machines.

4. QoS Requirements: Meeting the QoS requirements during network communication in terms of

    - Packet Rate;
    - Packet Order; and
    - Latencies associated with message transmissions.

5. Fault Tolerance: Providing reliability and fault tolerance mechanisms within the system

System Shutdown: CBWare achieves graceful termination of each process by looking up a lock file generated on each machine during system startup. This file contains the process ids of all the running processes on each machine associated with the system. All the processes that have their process ids listed in the lock file on each machine are terminated. The processes are terminated on each machine in the reverse order in which they were started on each machine.

Synchronization: It is essential to have the same version of programs and configuration files on all machines to avoid inconsistent behavior of the system. CBWare addresses this issue by creating a tarball of the entire system consisting of all the programs and the configuration files. This tarball is propagated to all the machines at system startup time. Whenever a change is made to the system, a synchronization check is performed on all machines during startup and the change is propagated to all machines.

The ways in which CBWare addresses the other issues listed above are discussed in Chapters 4, 5, and 6.

4 CBWare

This chapter provides an overview of CBWare and discusses in detail the design and implementation of the different components of CBWare. It describes the data marshaling routines used to encode and decode the network data. It also highlights the specific features of CBWare that provided for easy debugging and tuning of algorithmic parameters.

4.1 Overview of CBWare

CBWare, which works on the Publish/Subscribe model, is a package for communication between distributed programs, where the producers and consumers of data are independent of each other. One of the main design principles of CBWare was that, except for the properties of data written to or read from CBWare, a module in the system does not need to know anything about the module that has generated or will consume the data. This decoupling of modules is central to achieving the design criteria mentioned above.

CBWare provides two types of interfaces. A typed queue interface, CBQueues, for reading and writing messages, and a typed message packet interface, CBPackets, for only writing messages. CBQueues provides distributed queues using a combination of shared memory and UDP communication. The data written to the queues are distributed to other computers using a UDP broadcast. One of the modules of the CBWare called the “cb_logd” supports logging data to disk and also broadcasts data on the wireless network for remote real-time monitoring of the system. CBWare enables transmitting data to heterogeneous environments using the CBQueues interface with a negligible transmission time. Every message type has a specialized data marshaling routine that deals with byte ordering and endian issues.

Figure 4 depicts the CBWare architecture with the various modules of the software

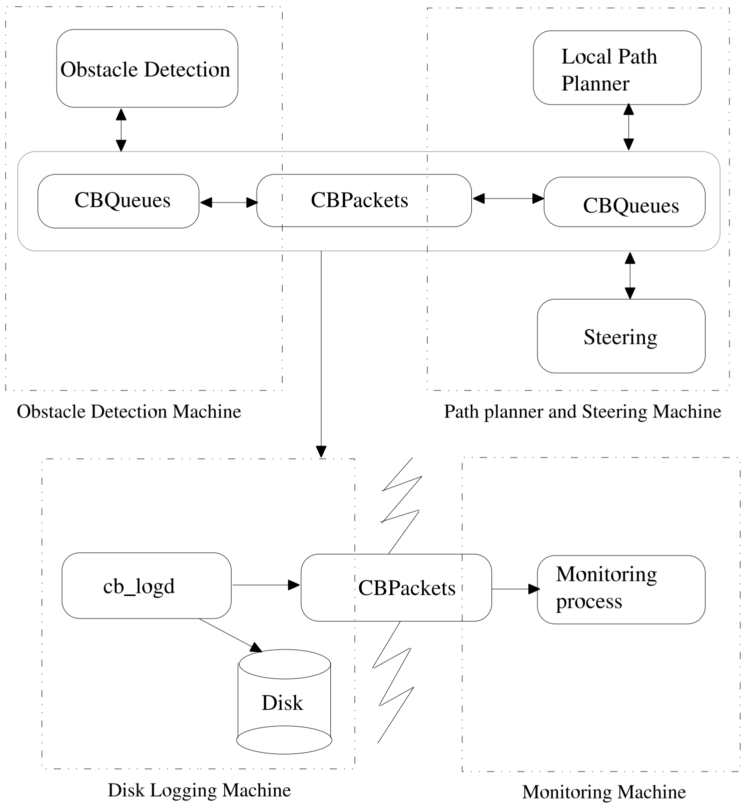

Figure 4: CBWare Architecture

system distributed on multiple machines. As shown in Figure 4, the local path planner and the steering modules running on “Path Planner and Steering Machine” communicate with each other through shared memory using the CBQueues interface. Distributed interprocess communication is achieved by replicating the shared memory queues of “Obstacle Detection Machine” on “Path Planner and Steering Machine” through UDP broadcast using the CBPackets interface. The “cb_logd” module running on “Disk Logging Machine” receives all the data from the UDP broadcast and writes the data to disk. It also transmits the logged data on a wireless network to the monitoring process running on “Monitoring Machine”(typically a laptop).

4.2 CBQueues

This section discusses the concept of the Shared Memory model for communication among processes and the advantages of using shared memory for inter process communication. It also gives insights on the design of shared memory used in the CajunBot software system and highlights the specific features of how CBQueues contributed to increasing the visibility of CajunBot by facilitating fusion of mutually consistent data from various sensors.

4.2.1 The Shared Memory Model

The term “shared memory” refers to a designated area of the memory that will be used simultaneously by multiple processes in a machine. This means that the data present in the shared memory area will be shared by several processes and all these processes can modify the contents of the shared memory area. Figure 5 shows the memory layout of three processes. Each process has its own code and private data area.

The shared data area as shown in Figure 5 appear to each process as its own address space. However, the area is shared. If one process writes into the shared data area, the change

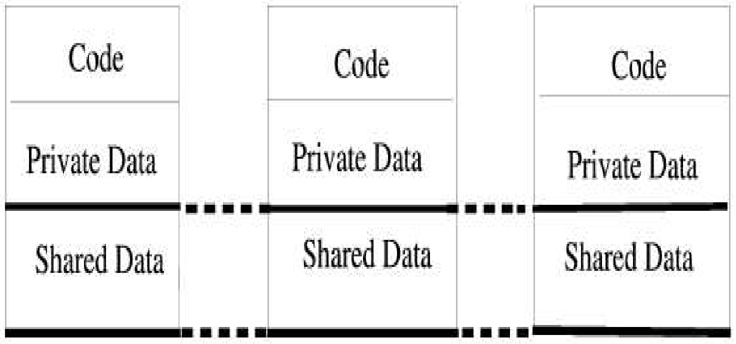

Figure 5: Multiple Processes Sharing Memory

is reflected in the shared space of the other processes. The shared memory area is indistinguishable to a process from its own (unshared) memory, except for some initializations needed to create the shared memory.

4.2.2 Message Queues

The software system of CajunBot consists of several concurrent processes communicating with each other on a single machine as well as over the network.

The Interprocess Communication on a single machine was built using POSIX Shared Memory [16]. The Shared Memory model was chosen for interprocess communication on a single machine in CajunBot due to the fact that shared memory provides an efficient mechanism to transfer data between processes on the same machine since its communication overhead is negligible. CajunBot consists of a large amount of data generated from various sensors and the data needs to be delivered to other processes in a timely manner. Therefore, we use the speed of shared memory to deliver the high bandwidth data from one process to another on an individual machine. The network is used to distribute the data to multiple machines. Though the Shared Memory model has negligible communication cost, there should be mechanisms to synchronize access to the shared memory to avoid creating race conditions and inconsistent states.

The data that needs to be shared by processes on a single machine is maintained as a queue of data for each message type in shared memory. The queues are maintained as circular lists in shared memory. Producers and Consumers write/read using the CBQueues interface. The queues are implemented using a fixed size array whose space is allocated during initialization of the shared memory. This method of shared memory is efficient because it allocates all the space that it will ever use at initialization time. The queues' buffer space is allocated once and is simply reused over and over again rather than being dynamically allocated as needed.

4.2.3 Message Format

All the message types of the CajunBot software system have been defined using the "struct" format of the C language. In addition to containing information pertaining to the respective messages, each message has two fields in common, the message name and the time stamp. The message name is used to determine the type of the message. Every message is time stamped before publishing the message to the queue. Time stamping every message is crucial for the data interpolation feature provided by the CBQueues. The interpolation feature is explained in Section 4.2.5.

4.2.4 CBQueues Interface

CBQueues synchronizes access to the shared memory using the following conventions.

1. Each queue can have only one writer, but there is no limit on the number of readers (Single Writer Restriction).

2. Each reader maintains its own point in the queue.

3. Each queue is made long enough such that a fast producer (writer) and a slow reader (consumer) can function together, without the producer clobbering the memory space that a reader is still reading. This may, however, require that the reader read the most recently written data, not the record after the most recently read.

The "Single Writer (Producer) Restriction" ensures that when the shared memory is replicated on other machines, data in each distributed queue can be temporally ordered on the time the data were produced. If multiple producers of similar type of data exist, such as multiple LIDARs, a separate queue is maintained for each producer.

CBQueues provide the following interfaces to access a message queue.

- **Reading the most recent message in the queue:** This interface provides the message at the head of the queue which is also the most recently written message. This method will block if there is no message at the head, which may occur if no writer has been created for the queue or if the writer has not placed any message in the queue at all.

- **Reading the next message from queue:** This interface provides the subsequent message in the queue, if available. If there is no next message, because the reader has already read the most recent message, then this interface blocks, waiting until a new message is added in the queue.

- **Checking for new message in queue:** To perform an unblocked read of the next message, the reader may want to check if it is already at the head. If the reader is at the head of the queue, then there is no message to read.

- **Writing a message in the queue:** This interface appends a message to the specified queue.

4.2.5 Interpolation of Data

Besides providing the usual interfaces to access a queue as described in the Section 4.2.4, CBQueues also provides an interface to find in a queue two data items produced around a particular time. This capability, made possible due to temporal ordering of data in the queues, provides support for fusion of data from multiple sources based on the time of production. When two sources generate data at different frequencies, it may not always be appropriate to use the most recent data from both sources. Doing so may lead to the fusion of mutually inconsistent data. For instance, when a LIDAR scan is mapped to global coordinates using the INS data, the resulting coordinates would have significant error if the vehicle experienced a sharp bump immediately after the scan. In such cases, it is better to fuse data in close temporal proximity. Along the same lines, instead of using the data generated directly by a source, sometimes it is preferred to interpolate the data for the specific time when data from another source are produced. In our LIDAR and INS example, it may be preferred to interpolate the position of the vehicle to match the time of LIDAR scan.

4.2.6 Utilities for CBQueues

The following utilities have been provided to work with CBQueues and the shared memory.

- **Remove a corrupted shared memory:** Most often when the shared memory is corrupted, it causes the programs to crash. The problem often goes away after cleaning of the shared memory. "cb_qclean" is a utility that is used to clean up the shared memory.

- **Read and write to shared memory, for testing:** This utility is useful in terms of testing and debugging problems with the queues in shared memory. It has provisions to read data from a specific queue in shared memory and also to write to a specific queue in shared memory. Some of the other options that this utility provides are to display binary output from the queue reader, to have a delay between every successive write operation to the queues, and an "interp" option to interpolate two data items produced around a particular time.

4.3 CBPackets

This section provides insight on the design choices that were used to implement CBPackets. It also discusses in detail on how CBPackets is used to achieve distributed Interprocess Communication.

4.3.1 The Communication Protocol

Transmission Control Protocol (TCP) and User Datagram protocol (UDP) are the two most commonly used transport layer protocols for network data transfer between processes on different hosts. TCP is a connection-oriented protocol that guarantees reliable and in-order data transfer. UDP is a connectionless protocol that does not guarantee reliability and ordering of packets. UDP is faster because it does not have the overhead of checking if every packet arrived and retransmitting lost packets, if any. It is primarily used for time-sensitive applications where receiving most of the data is more important that receiving all of the data in order. In systems that involve one-to-many or many-to-many interactions, UDP is required because of its broadcast and multicast capabilities.

Scenarios in robotic applications where UDP can be used as the communication protocol are listed below.

- While developing software for an autonomous vehicle, many of the core modules of the software system running on multiple machines require the global positioning data, the speed data and the heading data from the Inertial Navigation System (INS) for performing computations. Hence, broadcasting the INS data through the UDP protocol would be an efficient mechanism to transfer data over the network with negligible communication overhead rather than establishing point-to-point TCP connections with every machine in the system. TCP connections with each machine add a lot of overhead in establishing connections and transferring data.

- In most of the autonomous vehicles, usually one of the processes on the vehicle sends noncritical status information to a remote monitoring machine through a wireless network. In such cases, if the status information is sent using TCP, there might be problems if the wireless network fails. The process tries to re-send the information, because of the reliability nature of the TCP protocol. So, in such cases, where the information that is sent over the network is used only for monitoring, UDP would solve the problem of the wireless network failure since UDP is a connectionless protocol and never re-sends the messages.

We make a suggestion for designing the system architecture to overcome the reliability issue involved with UDP. We suggest that processes that need to share critical information, for example emergency control signals, can be grouped together on the same machine and communicate using shared memory on a single machine. Hence, the data transfer becomes reliable.

4.3.2 CBPacket Format

The data broadcast over the network are in the form of UDP Packets. As shown in Figure 6, the CBPacket consists of a header and the data that needs to be transferred across the network. The packet header is constructed by the CBWare before sending the packet over the network.

This header contains information pertaining to the data.

The header consists of the following fields.

1. **Channel:** The channel is a 4-byte field, which is a unique integer associated with each message type to ensure consistency in resolving the type of message on all machines. The message type is a combination of a unique message ID and a message name. The mapping of a channel to a message type is done through a map file that consists of the channel and the message type. All the machines in the system have the same version of the map file to avoid ambiguities in resolving the type of the messages.

2. **Tstamp:** This 8-byte field consists of the time at which the packet is sent over the network. Having the Tstamp in the header is useful in two ways: first, to check for any out of order packets, second, to calculate the delay associated with every packet while traveling from the source to the destination.

3. **Size:** This 4-byte field consists of the size of the data transferred over the network. The maximum size of a UDP packet that can be sent over the network is 64K.

4. **Checksum:** The checksum is a 4-byte field to verify the integrity of the data transmitted over the network. The checksum is a value computed based on the contents of the data and varies for every packet. This computed checksum is stored in the header and sent along with each packet. The receiver of the data computes a new checksum of the received data using the same procedure as in the sending side. The newly computed checksum on the receiving side is compared to the one sent with the packet. If both these values are the same, there is a high probability that the data was not corrupted. In CBWare, the checksum is calculated by performing an XOR operation of every byte in the data.

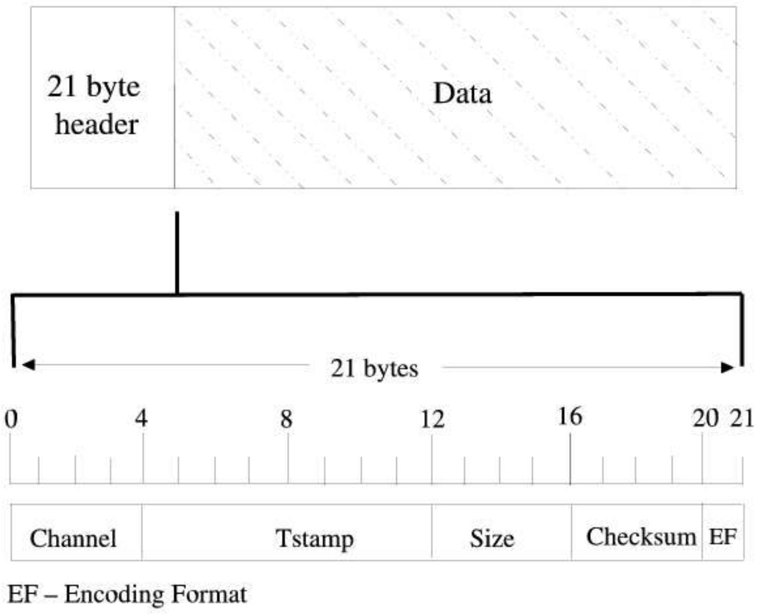

Figure 6: CBPacket Format

5. **Encoding format:** The encoding format is a byte that is stored in the header to determine the type of data being transmitted over the network. For example, the raw data from INS are just a sequence of bytes, which does not have any fixed structure. This type of raw data cannot be marshaled because of the varying size and format of the data. On the other hand, messages, which have a fixed format, need to be marshaled to a neutral format to be transferred across the network. The encoding format field in the header is used to identify the type of the data sent and received over the network.

The data portion of the packet is an array of "size" bytes (size of the data is stored in bytes in the header) where size < (62250 – (size of the header)) (See Figure 6).

A distributed system might contain processes running on heterogeneous platforms interacting with each other. In such scenarios, the packets that are sent over the network should be converted from the host format to the network format to deal with byte-ordering differences of the various machines in the system. There are various data encoding formats

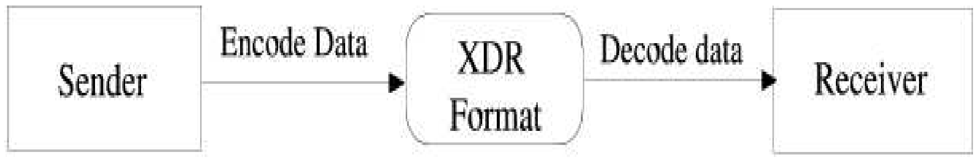

Figure 7: XDR Translation Operations

that are used to perform these conversions. Unix Sockets offers specific functions, “htons” and “ntohs,” that perform the host format to network byte-order conversion and vice-versa, respectively.

4.3.3 Data Marshaling

Data marshaling is the process of converting data from the local machine format (native format) into a standard format used for network transfer and retranslating the data received from the network back to the native form. In a distributed system, there might be processes running on different machines in heterogeneous platforms. These machines might follow different byte orderings and alignment strategies. For all these machines to be able to exchange data over the network, data need to be marshaled before being sent across the network.

The various data encoding standards that are used to convert data into a machine-independent format have been described in [17]. eXternal Data Representation (XDR) is a standard developed by Sun Microsystems [18] for encoding data. It is useful for transferring data between different computer architectures and has been used to communicate between diverse machines [19].

The advantage in using the XDR standard to represent data is that it has only one convention of marshaling the data. Therefore, data are encoded to the “XDR format” before being sent over the network, and at the receiving side, data are decoded back to the local machine format. This enables adding machines with new computer architectures in the system without having to modify any of the translation routines. There are two operations that need to be carried out to marshal the data as shown in Figure 7. The process that sends the data across the network needs to do an “encode” operation to convert the data into the XDR format and the process that receives the data from the network needs to perform a corresponding “decode” operation to convert the data to the native format supported by the machine. CBWare encodes the UDP Packet into XDR format before sending the packet across the network. On the receiving side, CBWare decodes the data into the local machine format for the processes running on that machine to use the data.

4.4 XDR Translation Procedure

XDR provides translation routines for primitive as well as constructed data types in C. A detailed description on all the translation routines provided by XDR can be found in [20].

A header translation routine has been written to encode and decode the header of the CBPacket using the routines provided by XDR. In the same manner, a message translation routine has been written for every message type that is transmitted across the network. Since XDR represents every data item as a multiple of four bytes, the size of the data in native format will be different from the size of the data in XDR format. Hence, along with writing a translation routine for every message type, there should be a corresponding routine written for every message that calculates the size of the message in XDR format.

The steps involved in the translation (encoding or decoding) of the CBPackets are as follows.

1. An XDR buffer is created that is large enough to hold the entire CBPacket, and the direction of translation (encoding or decoding) is specified while creating the buffer.

This direction of translation is stored as one of the attributes of the XDR stream.

2. The header is marshaled and added to the XDR buffer using the header translation routine.

3. The message is marshaled and added to the XDR buffer using the message translation routine.

The XDR buffer contains the marshaled header followed by the marshaled data. Based on the direction of translation in the XDR buffer, further operations are carried out with the buffer. The appendix presents the translation routines that have been written for the "scan" message (data generated from the LIDAR's) of CajunBot and also the routines that calculate the size of the data in XDR format. Whenever a new message is added to the system, a corresponding message translation routine and a routine that calculates the size of the message in XDR format should be written for the newly added message.

4.5 Log System of CBWare

The log system of CBWare provides the following features.

1. It maintains a mapping between each message type and a unique integer resulting in a logical channel for each message type.

2. There might be cases when a particular message type might be broadcast by more than one process, running on different machines, which may cause network congestion due to duplication of the same message being broadcast from multiple machines. Based on the information contained in the log system, each process running on every machine decides whether to broadcast a particular message type. The log system is defined in

such a way that no two machines broadcast the same message on the network. This avoids redundant data being broadcast on the network.

3. When the system runs for long hours, there is a high probability that the disk to which data is being logged might run out of disk space. To overcome this problem, the log system has provisions to save only a sample of the data produced and broadcast on the network. The log frequency for each message type varies based on the importance of the message in debugging and post-analysis.

4. The remote real-time monitoring of CajunBot happens through a wireless network. The variety of the wireless equipment we have tried tends to crash when all the data produced in the queues on all the machines are sent on the wireless network. To accommodate for the shortcomings of the wireless network, the log system facilitates sending only a sample of the data produced in the queues on the wireless network. The frequency of data sent on the wireless network depends on the importance of the data in terms of real-time visualization.

   The log system of CBWare maintains all the information listed above in a file that is consistent across all the machines in the system.

4.6 Sending the Messages

Every process apart from writing data to the CBQueues on a machine also uses CBWare to publish data on the network. “cb_publisher,” which is the publisher component of the CBWare, handles the chores of establishing the UDP network connection through low-level network programming using sockets, constructing the CBPacket, marshaling the packet into the XDR format and sending the message in the form of CBPackets across the network.

4.7 Central Log Server

The data written to the shared memory queues on every machine should also be logged to disk for post-analysis and debugging purposes. A central log server is maintained that receives all the data from the onboard broadcast network and logs the received data to disk. “cb_logd,” which is the central log server, is a daemon that runs on a separate machine as shown in Figure 4. It receives all the data from the onboard computers and writes data pertaining to each message on separate files in the disk. The data written to each file is in binary format. The logging operation has been moved to a separate machine so that a disk failure, a very likely possibility in a 10-hour run, does not interfere with the autonomous operations of the vehicle. “cb_logd” logs only a sample of the data produced by the onboard processes, if needed, through the information provided by the log system. Besides logging the data to disk, “cb_logd” also samples the data at specific intervals as per the information contained in the log system and broadcasts the data on the wireless network for real-time monitoring.

4.8 Receiving the Messages

“cb_subscriber,” which is the subscriber component of CBWare, is a daemon that runs on each machine in the system and receives the broadcast data sent by the “cb_publisher.”

“cb_subscriber” runs in two modes.

1. **Mode 1:** When “cb_subscriber” runs on the onboard computers, it receives the messages from the onboard broadcast network. “cb_subscriber” only subscribes to and receives messages that are necessary for the processes running on the respective machines. It discards the other messages that are not required by the processes on each machine.

2. **Mode 2:** When “cb_publisher” runs on the laptop used for remote real-time monitoring,

it subscribes to and receives all the messages sent on the wireless network by the "cb_logd" process.

"cb_subscriber" receives the UDP packets from the network in the encoded form. Based on the encoding format information present in the header of the packet, the "cb_subscriber" decodes the message appropriately into the local machine format. After decoding,"cb_subscriber" writes the message to the corresponding message type queue in the local shared memory of the machine using the CBQueues interface, which then can be used by processes local to that machine. Thus, we replicate the shared memory of one machine on another. This feature of CBWare to distribute queues over other machines allowed easy transfer of programs over multiple machines, achieving easy scalability of computational power, one of the design criteria.

The onboard broadcast address and the wireless broadcast address on which messages are sent and received is a combination of an IP Address and a port number. The IP Address is the broadcast address of the respective networks. The port number is a numeric identifier used to talk to a specific process in the system. These addresses are maintained as environment variables that are consistent throughout the system.

4.9 Monitoring Process Status Information

When there are many processes distributed across multiple machines in a system, each of these processes generates several status, warning, or error messages (CbMessages) on their respective machines. These messages are difficult to track down on each individual machine in real-time and doing so becomes a tedious task, especially when the system scales to a higher level. All the message should be logged to the disk for later analysis to identify problems with the system. To facilitate centralized monitoring and logging of CbMessages from each process on all machines, CBWare has a special component, the "cbmesg_publisher," which handles the messages generated by each process. Every process uses the “cbmesg_publisher” to publish the CbMessages on the broadcast network just like any other message type in the system. The only difference with the CbMessages type is that they are not written to the shared memory queue before publishing the messages on the network. This is due to the single writer restriction imposed by the CBQueues as discussed in Section 4.2.4. There may be multiple processes that produce CbMessages, so the single writer restriction does not hold for CbMessages. The contents of the CbMessage type are listed below.

- **Tstamp:** This field denotes the time at which the CbMessage was generated.
- **Machine Name:** Since there are multiple machines that may be generating CbMessages, this field stores the machine name from which the message originates.
- **Program Name:** This field denotes the program that generates the CbMessages.
- **CbMessage:** This field contains the CbMessage.
- **Type:** This field indicates the type of the CbMessage that was generated, and it can be one of four types, as listed below:
    1. Error Message;
    2. Status Message;
    3. Warning Message; or
    4. Perror Message.

The CBPackets interface provides support for multiple writers and multiple readers. However, in so doing it cannot support temporal fusion of data. This interface is used only for logging and real-time monitoring of status, warning, and error messages. Such messages are used in isolation, that is, they are not fused with other messages, and are mostly used for monitoring, not control.

4.10 Log Control

CBWare has provisions to remotely control the logging process by sending control signals to the "cb_logd" daemon. This control process connects to the logging process using TCP and sends control signals using a predefined ASCII Protocol.

The control signals can be one of three types, that

1. Enable logging of data;
2. Disable logging of data; or
3. Change the directory on the logging machine in which the data is currently being logged to the directory specified in the control signal.

To send these signals, the control process needs to know the IP Address of the log daemon and the port number on which the log daemon is listening for the control signals. The IP Address and the port number are maintained using an environment variable in the system.

5 CBWare Performance

Evaluation of middlewares based on QoS metrics guides the development of middlewares, the design of system architecture, and network configuration in distributed system applications. There have been many surveys on QoS metrics for evaluating middlewares, especially for publish/subscribe middlewares [21], [22]. The most commonly used metrics for evaluation of middlewares in all these surveys were message delivery guarantee, timely delivery, and security.

This chapter evaluates the CBWare on the following QoS metrics:

- End-to-End Latency;
- Packet Rate;
- Bandwidth; and
- Packet order.

These metrics are critical and important performance measures for evaluating middlewares developed for real-time distributed applications.

5.1 Performance Measure of CBWare using QoS Metrics

End-to-End Latency: This is the one-way delay that is associated with a single message from the time it is sent by the publisher until the time the message is received by the subscriber. This parameter is also known as the transmission delay.
Packet Rate: Rate refers to the number of packets of a single message that can be transferred through the network in one second.

Bandwidth: Bandwidth is the data rate that is supported by the network. Typically, it refers to the amount of data that can be transmitted through the network at any given point in time. It is normally measured in bytes per second. Bandwidth is also knows as throughput.

For any message that is transmitted across the network under any given condition, the following relation holds true.

$$PacketRate = Bandwidth/MessageSize \tag{1}$$

Packet Order: Order of packets is the order in which the packets of a message arrive at the destination. While using UDP, packets may arrive out of order. For example, packet A may be sent before packet B, but B arrives before A at the destination. The Packet Order metric is used to determine whether there were any out-of-order packets for a given message transmitted through the network.

We determine a set of QoS requirements in terms of the maximum values allowed for all the above mentioned metrics which is the limit that is tolerable by the system and that does not affect the autonomous operations of the vehicle.

5.1.1 Experimental Setup

All the experiments shown below were done on the autonomous vehicle with Dell Poweredge 750 servers onboard connected using a 1 GB Ethernet LAN Switch. The results may vary based on the type of computers and network configuration.

5.1.2 End-to-End Latency Measurement

We calculate the end-to-end latency for five messages that were transmitted across the network, with widely varying sizes (in bytes) ranging up to 62000 bytes. The results show

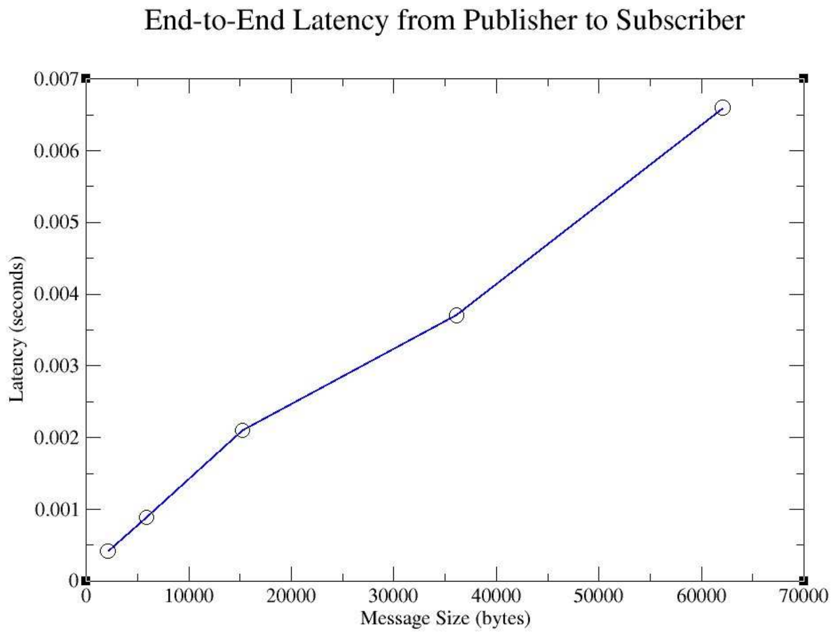

Figure 8: End-to-End Latency

that the latency increases as the message size increases. The transmission delay ranges from 0.4 msec to 6 msec for message sizes ranging from 2000 bytes to 62000 bytes, respectively, as shown in Figure 8. We specify a maximum tolerable transmission delay of 5 msec.

The subscriber of messages on each machine, "cb_subscriber," calculates the difference between the time in the header of the packet and the packet reception time. The difference is termed as the transmission delay. The time in the CBPacket header is the time at which the data was sent across the network. If the transmission delay is over 5 msec, then the subscriber raises a "cbmesg" alarm (warning CbMessage) for the specific message type that can be monitored by the monitoring process on the laptop through the wireless network.

5.1.3 Bandwidth and Packet Rate Measurement

Table 1 combines the measurements for Bandwidth and Packet rate and also shows the relation between Bandwidth, Packet Size, and Message Size according to Equation 1. This

Table 1: Relation between Message Size, Bandwidth, and Packet Rate

Message Size (bytes)	Packet Rate	BandWidth (bytes/second)
2112	1200	2534400
5860	475	2783500
15236	195	2971020
36084	84	3031056

measure indicates that the maximum number of packets supported by CBWare in one second ranges from 84 packets/second to 1200 packets/second, which is heavily dependent on the message size.

5.1.4 Packet Order

We determine if there were any packets that arrived out of order for every message. The ordering is determined by the "cb_subscriber" which compares the publish timestamp (the timestamp in the CBPacket header) of every received packet of a message (current timestamp) with the publish timestamp of the previously received packet of the same message (last timestamp). If the current timestamp < last timestamp, then we find an out of ordering of packets and discard the packet that arrived out of order. The "cb_subscriber" also raises a warning message indicating that there was an out-of-order packet that was received. One of the main reasons for packets arriving out of order is due to the high levels of jitter experienced by the network. Our experiments indicate that "cb_subscriber" did not report any out-of-order packets, which leads us to believe that the onboard network configuration on CajunBot, which is the 1 GB Ethernet LAN used by CBWare to transmit messages, did not experience jitters or network congestion.

However, the remote monitoring process that runs on the laptop connected through a wireless network reported many out-of-order packets for every message. Even though the messages that are sent on the wireless network are sampled at a regular interval, the variety of wireless equipment we have tends to experience high jitters and network congestion when there are a huge number of messages transmitted from the onboard computers across the wireless network. In order to increase the efficiency of the wireless access point and the strength of the wireless signal, we added a wireless signal booster, after which the number of packets that arrived out of order was reduced by a considerable amount.

5.2 CBWare's Network Performance Measure Using Ethereal

We used a network monitoring tool, Ethereal [23], to monitor the live network traffic and analyze the packets that are sent across the network in real-time. Ethereal has a graphical user interface and offers a variety of filters, for example, to capture data that is sent or received only on a particular network interface and network port or to capture only UDP traffic across a network. Ethereal also provides a "Statistics: Summary" option to display the summary information about a recent capture. We used the Ethereal "Statistics: Summary" option to measure the packet rate and bandwidth metrics described in Section 5.1.3. Ethereal also has provisions to capture live data and store the data in a file, which can be used for post-analysis.

The experiments and the measurements of the QoS metrics helped us evaluate the performance of CBWare with respect to the distributed system architecture and network configuration and suggested potential improvements that could be made in future.

6 Conclusions and Future Work

This thesis developed a middleware for an unmanned autonomous ground vehicle. The middleware, CBWare, facilitated a transparent communication infrastructure for the different processes in the software system of the autonomous vehicle to exchange information in a distributed environment satisfying the QoS requirements. CBWare provides specialized support for fusing mutually consistent data from multiple sensors present in the autonomous vehicle. CBWare with its real-time remote monitoring capability has proven to be an excellent debugging tool in terms of analyzing the behavior of the autonomous vehicle and tuning algorithmic parameters in real-time. CBWare was developed using a combination of the shared memory model and network-based UDP broadcast mechanism for communication among processes running on heterogeneous environments in the distributed system. It works on the Publish/Subscribe model of communication. The shared memory model was used for communication among processes running on a single machine. The shared memory on one machine was distributed to other machines using UDP Broadcast.

The middleware was used in CajunBot, a finalist in the DARPA Grand Challenge 2005. The middleware was evaluated on various QoS metrics like end-to-end transmission delays, packet rate, and a few others. The results show that there was negligible communication overhead involved in transferring messages from one machine to another.

Currently CBWare does not provide a fault-tolerance mechanism when any of the components of the autonomous vehicle fail to function in the required manner. The fault-tolerance mechanism involves a watchdog process that monitors all the machines in the system, replicates components at various levels in the system in case of failure, and transfers processes running on one machine to other machines when the machine fails. These features enable the system to operate in the specified manner even when there are failures. The fault-tolerance mechanism would be an important addition to CBWare in the future, which would prevent the system from going down in case of failure.

The maximum packet size supported by the UDP Protocol is 65535 bytes (64k). In our current software system of CajunBot, we do not have any messages that are greater than 64k. In the future, when messages greater than 64k need to be sent across the network, it would be useful to have a good compression mechanism for the messages or to break down the messages into smaller packets, transmit them across the network, and reassemble them at the receiving side. However, the entire message becomes invalid if even one of these smaller packets is not delivered to the destination. It would also be worth investigating the overhead involved in breaking down the messages into smaller packets and reassembling them at the destination.

CBWare currently supports Fedora Core 2.0 and 3.0. We are exploring possibilities and changes that need to be made to port CBWare to other operating systems. In our attempts to support CBWare on Windows, we have found out that we cannot port CBWare's current model of shared memory from Fedora Core to the Windows Operating System.

References

[1] “DARPA Grand Challenge,” http://www.darpa.mil/grandchallenge (last accessed August 15, 2006).

[2] A.Virgillito, “Publish/subscribe Communication Systems: From Models to Applications,” Doctoral dissertation, Universit‘a degli Studi di Roma “La Sapienza”, 2003.

[3] P.Eugster, P.Felber, R.Guerraoui, and A.Kermarrec, “The Many Faces of Publish Subscribe,” *ACM Computing Surveys*, vol. 35, June 2003, pp. 114–131.

[4] A. D. Birrell and B. J. Nelson, “Implementing Remote Procedure Calls,” *ACM Transactions on Computer Systems*, vol. 2, no. 1, Feb 1984, pp. 39–59.

[5] J.Gowdy, “IPT: An Object Oriented Toolkit for Interprocess Communication,” Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-96-07, Mar 1996.

[6] O.Firschien and T.M.Strat, eds., *Reconnaissance, Surveillance and Target Acquisition for the Unmanned Ground Vehicle: Providing Survelliance Eyes for an Autonomous Vehicle*, Morgan Kaufmann Publishers, 1997.

[7] R. Simmons and D. James, *IPC – A Reference Manual, version 3.6*, Robotics Institute, Carnegie Mellon University, Aug 2001, http://www.cs.cmu.edu/afs/cs/project/TCA/ftp/IPC_Manual.pdf (last accessed September 2, 2006).

[8] J. Pederson, "Robust Communication for High Bandwidth Real-Time Systems," Master's thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 1998.

[9] S.Schneider, G.Pardo-Castellote, and M.Hamilton, "NDDS: The Real-Time Publish-Subscribe Middleware," Real-Time Innovations Inc., Tech. Rep., Aug 1999.

[10] W. P. Shackleford, F. M. Proctor, and J. L. Michaloski, "The Neutral Message Language: A Model and Method for Message Passing in Heterogeneous Environments," *Proceedings of the World Automation Conference*, Maui, Hawaii, June 2000.

[11] H.Utz, S.Sablatnog, S.Enderle, and G.Kraetzschmar, "Miro-Middleware for Mobile Robot Applications," *IEEE Transactions on Robotics and Automation*, vol. 18, Aug 2002, pp. 493–497.

[12] T. Harrison, D. Levine, and D. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," *Proceedings of OOPSLA*, Atlanta, Oct 1997.

[13] M.McNaughton, S.Verret, and H.Zhang, "Broker: An Interprocess Communication Solution for Multi-Robot Systems," *IEEE International Conference on Intelligent Robots and Systems (IROS)*, Edmonton, Canada, Aug 2005, pp. 1458–1463.

[14] J.Gowdy, "A Qualitative Comparison of Interprocess Communication Toolkits for Robotics," Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-00-16, June 2000.

[15] M. Matteucci, "Publish/subscribe Middleware for Robotics: Requirements and State of the Art," Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy, Tech. Rep. Technical Report N 2003.3, 2003.

[16] A. D. Marshall, *Programming in C, Unix System Calls and Subroutines using C*, 1999, ch. IPC: Shared Memory, http://www.cs.cf.ac.uk/Dave/C/node27.html (last accessed August 25, 2006).

[17] G. Sadavisam and A.Chitra, "Certain Improvements In Marshalling," *Academic Open Internet Journal*, no. 11, Feb 2004.

[18] "XDR: External data representation standard," Sun Microsystems Inc., Tech. Rep. RFC 1014, June 1987.

[19] *Network Programming Guide - External Data Representation Standard: Protocol Specification*, Sun Microsystems Inc., 1990.

[20] "Library routines for xdr data representation," http://docs.sun.com/app/docs/doc/806-7022/6jfu47jvs?a=view (last accessed September 10, 2006).

[21] S. Behnel, L. Fiege, and G. Muhl, "On Quality-of-Service and Publish-Subscribe," *Fifth International Workshop on Distributed Event-Based Systems (DEBS'06)*, Lisbon, Portugal, July 2006.

[22] A. Corsaro, L. Querzoni, S. Scipioni, S. T. Piergiovanni, and A. Virgillito, *Quality of Service in Publish/Subscribe Middleware*, IOS Press, 2006.

[23] "Ethereal: Network Protocol Analyzer," http://www.ethereal.com/docs/eug_html/ (last accessed September 15, 2006).

Appendix

This appendix describes the XDR translation routines written for the messages used in the CajunBot software system. Typically, every time a new message is added to the system, the designer of the message has to write the translation routine corresponding to the message using the XDR primitives. Since XDR treats every data item in the message as a multiple 4 bytes, there should be a routine, which calculates the size of the message in XDR format. Hence, along with the XDR translation routine, an "xdr_sizeof_datatype" routine also needs to be written for every message in the system.

Every message in the CajunBot software system is defined using structures in the C programming language. In this appendix, I describe the "scan" message, which contains data generated by the LIDAR sensors, their respective XDR translation routines, and the corresponding "xdr_sizeof_datatype" routines.

The scan message

The scan message contains information from the onboard LIDAR sensors in the autonomous vehicle. The scan message has been defined in the CajunBot system as follows.

```
struct scan_data_t
{
    static char const * const NAME;
    double tstamp;
    unsigned status;
    unsigned num_beams;
    scan_beam_t beam[SCAN_MAX_BEAMS];
};
```

```
struct scan_beam_t
{
    float range;
    float theta;
};
```

The XDR Translation routines

For each of the above structures defined for representing the scan message, an XDR translation routine has been written to marshal the data in XDR format. The XDR translation routine translates each field in the structure to its corresponding External representation form and does the vice-versa operation depending upon the direction of translation. The following procedures are the XDR translation routines for the scan message type.

```
unsigned xdr_translate_primitive (XDR *xdr, scan_data_t *d)
{
    if (xdr_double (xdr, &d->tstamp)
    && xdr_u_int (xdr, &d->status)
    && xdr_u_int (xdr, &d->num_beams)
    && xdr_vector (xdr, (char *)d->beam, SCAN_MAX_BEAMS,
    sizeof (scan_beam_t), (xdrproc_t)
    xdr_translate_scan_beam))
        return xdr_getpos (xdr);
    return 0;
}
```

```
    bool xdr_translate_scan_beam (XDR *xdr, scan_beam_t *d_)
    {
         if (xdr_float (xdr, &d_->range)
         && xdr_float (xdr, &d_->theta))
              return true;
         return false;
    }
```

The "xdr_sizeof_datatype" routine

```
    unsigned xdr_sizeof_datatype (scan_beam_t *d)
    {
         return (xdr_sizeof ((xdrproc_t) xdr_float, &d->range)
         + xdr_sizeof ((xdrproc_t) xdr_float, &d->theta));
    }

    unsigned xdr_sizeof_datatype (scan_data_t *d)
    {
         return (xdr_sizeof ((xdrproc_t) xdr_double,
                             &d->tstamp) +
         xdr_sizeof ((xdrproc_t) xdr_u_int, &d->status) +
         xdr_sizeof ((xdrproc_t) xdr_u_int, &d->num_beams) +
         (SCAN_MAX_BEAMS * xdr_sizeof_datatype (d->beam)));
}
```

Table 2: Description of the XDR primitives used in scan message type

XDR Primitives	Description
`xdr_double`	Translates between a double precision variable, and its corresponding external representation
`xdr_u_int`	Translates between an unsigned integer variable, and its corresponding external representation
`xdr_vector`	Translates between a fixed length array, and its corresponding external representation
`xdr_float`	Translates between a float variable, and its corresponding external representation
`xdr_getpos`	Gives the current position in the xdr translation buffer
`xdr_sizeof`	Gives the number of bytes required to encode a variable

The “xdr_sizeof” primitive returns the number of bytes required to encode the data. The “xdr_sizeof_datatype” routines that have been written for the scan message type calculate the total number of bytes that are required to encode/decode the scan message by applying the “xdr_sizeof” primitive to each field in the scan message structure.

Table 2 lists each of the XDR primitives and a brief description about each primitive used in the XDR translation routines and the “xdr_sizeof” routine for the scan message type.

Venkitakrishnan, Vidhyalakshmi. Master of Science Technology, Birla Institute of Technology and Science, Fall 2004; Master of Science, University of Louisiana at Lafayette, Fall 2006
Major: Computer Science
Title of Thesis: CBWare - Distributed Middleware for Autonomous Ground Vehicles
Thesis Director: Dr. Arun Lakhotia
Pages in Thesis: 63; Words in Abstract: 239

ABSTRACT

Distributed Real-time Systems are playing a crucial role in many application domains, especially in robotics. Needs arise in robotics where components within a machine and components distributed across a network must exchange information seamlessly with minimal communication overhead, satisfying the Quality-Of-Service (QoS) requirements. The solution in this case is what is known as a “middleware” that acts as an intermediary between different application components in a distributed system.

This thesis presents CBWare, a network-based distributed message passing middleware developed for CajunBot, an unmanned autonomous ground vehicle. The middleware was developed with the objective of providing a transparent communication mechanism with real-time monitoring and debugging capabilities and minimal network communication latencies. CBWare provides specialized support for fusing data from various sensors arriving at varying frequencies and latencies.

The proposed middleware works on the Publish/Subscribe model such that the producers and consumers of data are independent of each other. Processes running on the same machine communicate using queues maintained in a shared memory area. Data written to the shared memory queues are distributed to other processes on multiple machines using a UDP Broadcast. The support for fusion of sensor data in CBWare is based on the time of production of data, thereby ensuring fusion of mutually consistent data. CBWare also provides capability to sample data at regular intervals to be transferred over the wireless network, thereby facilitating real-time monitoring and debugging capabilities.

Biographical Sketch

Vidhyalakshmi Venkitakrishnan was born in TamilNadu, India, on May 25, 1983. She graduated with a Master of Science Technology degree in Information Systems in June 2004 from Birla Institute of Technology and Science, Pilani, India. She entered the Master of Science program in Computer Science at the University of Louisiana at Lafayette in Fall 2004. Following completion of this degree, Vidhyalakshmi Venkitakrishnan is planning to pursue a Doctor of Philosophy in the area of Distributed Operating Systems.