CHAPTER 4

SIMULATION METHODOLOGY

This chapter describes the methodology used to design and implement the simulation to examine the performance of the proposed adaptive error control algorithm. First, a description of the PARSEC simulation language is given, followed by a discussion of the simulation environment which includes interaction process between entities, and the system parameters that are used throughout the simulations. Finally, the assumptions for this simulation are elaborated.

4.1 The PARSEC Language

PARSEC (PARallel Simulation Environment for Complex systems) is a C-based discrete-event simulation language developed at the Computer Science Department at UCLA. This simulation language adopts the process interaction approach to discrete-event simulation. An object (also referred to as a physical process) or set of objects in the physical system is represented by a logical process. Interactions among physical processes are modeled by time-stamped message exchanges among the corresponding logical processes.

The PARSEC language is derived from Maisie, but with several improvements both in the syntax of the language and in its execution environment. The PARSEC language is based on C, but introduces several new features. PARSEC programs consist of entities, which exchange messages each other.

4.1.1 PARSEC Entities

An entity definition describes a class of objects. An entity instance, henceforth referred to simply as an entity, represents a specific object in the physical system and may be created and destroyed dynamically. An entity is created by the execution of a new statement and is automatically assigned a unique identifier on creation. For instance, the following statement creates a new instance of a *manager* entity and stores its identifier in variable *r1*.

```
ename r1;
....
r1 = new manager (N);
```

An entity can reference its own identifier using the keyword **self**. PARSEC entity may terminate itself in any of the following ways: by executing a C return statement, by 'falling off the end' of the entity-body. All entities still active at the end of a simulation will be terminated by the runtime system. If an entity definition includes a **finalize** statement, the body of the **finalize** statement will be executed upon a normal termination of each instance of that entity type. The **finalize** statement is most useful for collecting the results of a program at its conclusion.

4.1.2 Message Communication between Entities

Simulation entities communicate with each other using buffered message passing. PARSEC defines a type called **message**, which is used to define the types of messages that may be received by an entity. Definition of a message-type is similar to a **struct**; the following declares a message-type called *request* with one parameter (or field) called count.

```
message request {int count;};
```

Every entity is associated with a unique message buffer. A message is deposited in the message buffer of an entity by executing a **send** statement. The send statement performs an asynchronous send: the sending entity copies the message parameters into a memory block, delivers the message to the underlying communication network, and resumes execution. The following statement will deposit a message of type *request* with time stamp *clock()+ t*, where clock is the current value of the simulation clock, in the message buffer of entity *m1*.

send *request* (2) **to** *m1* **after** t;

If the after clause is omitted, the message is time stamped with the current simulation time. If required, an appropriate hold statement may be executed to model message transmission times or a separate entity may be defined to simulate the transmission medium. An entity accepts messages from its message buffer by executing a **receive** statement. The receive statement consists of one or more resume clauses, and possibly a timeout clause. Each resume-clause consists of a read-only message variable, and an optimal **guard** followed by a statement. The timeout clause specifies a wait-time (t_c). If t_c is omitted, it is set to an arbitrarily large value. The resume-clause is a set of resume statements, each of which has the following form:

receive (m_t m_i) [**when** b_i] $statement_i$;

where m_t is a message-type, m_i is a read-only message variable, b_i an optional boolean expression referred to as a guard, and $statement_i$ is any C or PARSEC statement. The guard is a side-effect free boolean expression that may reference local variables or

message parameters. If omitted, the guard is assumed to be the constant true. The message-type and guard together are referred to as a resume condition. A resume condition with message-type m_t and guard b_t is said to be enabled if the message buffer contains a message of type m_t, which if delivered to the entity would cause b_t to evaluate to true; the corresponding message is called an enabling message. In general, the buffer may contain many enabling messages.

With the wait-time omitted, the wait statement is essentially a selective receive command that allows an entity to accept a particular message only when it is ready to process the message. For instance, the following receive statement consists of two resume statements. The resume condition in the first statement ensures that a *req* message is accepted only if the requested number of units are currently available (the requests are serviced in first-fit manner). The second resume statement accepts a free message:

```
receive (request req) when (req.count <= units)
        {
        or timeout in (free)
}
```

PARSEC also provides a number of pre-defined functions that may be used by an entity to inspect its message buffer. For instance, the function **qsize**(m_t) returns the number of messages of type m_t in the buffer. A special form of this function called **qempty**(m_t) is defined, which returns true if the buffer does not contain any messages of type m_t, and returns false otherwise. In general, the resume condition in a wait statement may include multiple message-types, each with its own boolean expression.

This allows many complex-enabling conditions to be expressed directly, without requiring the programmer to describe the buffering explicitly.

If two or more resume conditions in a **receive** statement are enabled, the time stamps on the corresponding enabling messages are compared and the message with the earliest time stamp is removed and delivered to the entity. If no resume condition is enabled, a timeout message is scheduled for the entity t_c time units in the future. The timeout message is canceled if the entity receives an enabling message prior to expiration of t_c; otherwise, the timeout message is sent to the entity on expiration of interval t_c. Thus the **receive** statement can be used to schedule conditional events. A **hold** statement is provided to unconditionally delay an entity for a specified simulation time. For instance, the statement **hold** (t) will suspend the corresponding entity for t units in simulation time.

4.1.3 The Driver Entity

Every PARSEC program must include an entity called driver. This entity serves a purpose similar to the main function of a C program. Execution of a PARSEC program is initiated by executing the first statement in the body of entity driver. The driver entity takes the same argc and argv parameters as the C main function (except that argv must be declared char** argv because of PARSEC's requirement that array parameters have a constant size). Parameters recognized by the PARSEC runtime system will be removed from argc and argv before the driver is invoked.

4.1.4 Entity Scheduling

In a PARSEC program, an arbitrary number of entities may be mapped to a single processor. The execution of these entities is interleaved by the PARSEC scheduler. Entities are scheduled for execution based on the timestamps of their enabling messages.

An entity can be in one of four states: idle, ready, active, or terminated. An entity that has been terminated does not participate any further in the program. An entity that has not been terminated is said to be idle if its message buffer does not contain any enabling message. An entity whose buffer contains an enabling message is said to be ready; at any given point, multiple entities on a processor may be in the ready state. The scheduler selects the ready entity with the earliest enabling message for execution which then becomes active. An active entity relinquishes control to the scheduler only if it is terminated or it executes a hold or receive statement. In the latter case, if its buffer contains an enabling message it transits to the ready state (and is hence eligible to become active immediately); if not, it transits to the idle state. It is important to note that an active entity is self-scheduled: the scheduler cannot force it to relinquish control. In particular, an active entity that never executes a receive (or hold) statement, will never relinquish control to the scheduler.

4.1.5 Other Features

The other features provided by PARSEC simulation language are program termination and clock operations. When a termination condition is detected in a program, each

entity's **finalize** statement will be called. The entity may take appropriate actions before termination, including printing accumulated statistical data. A PARSEC simulation terminates, whenever one of the following conditions arise:

1. The simulation clock exceeds the maximum simulation time specified by *setmaxclock()*.
2. All entities are suspended and no messages (including timeouts) are in transit.
3. An entity executes an *exit()* or *pc_exit()* statement.

Another feature is clock operations. This feature allow simulations to be executed over longer duration with fine grained clock values, the PARSEC system clock is implemented as a large integral type called **clocktype**. All clock operations make use of **clocktype** variables. The following functions are provided to manipulate the simulation clock:

1. *simclock(void):* This function returns the value of the current simulation clock as a **clocktype** value.
2. *setmaxclock(clocktype):* This function sets the maximum simulation time to the value specified in the numerical-string. The simulation is terminated when the simulation clock exceeds this value.
3. *atoc(char*, clocktype):* Places the **clocktype** value represented by the string in the **clocktype** parameter.
4. *ctoa(clocktype, char*):* Like **sprintf,** it prints the value of the **clocktype** parameter into the string parameter.

4.2 The Simulation Environment

As explained in the previous section, simulations were performed using the PARSEC simulation language. Another famous simulation language, Maisie is not used as a simulation environment because the PARSEC introduces several new features, both in the syntax of the language and in its execution environment. This simulation environment, allows for the creation of entities which operate in parallel and communicate via message passing. Furthermore, a common system clock is available to all of the entities in the simulation process.

4.2.1 Entities Interactions

For the purpose of this study, entities were defined to represent mobile hosts and communication channels. Briefly, there are five entities exist at one run time simulation: a *driver*, a *source*, a *sender*, a *channel*, and a *receiver*. The *receiver* entity can be either *error_ctrl*, *error_ctrl2* and *error_ctrl3* entities depend on user's selection parameter during program execution. These three different kinds of *receiver* entity represent the algorithms that user will choose at runtime simulation. The following is a description of the functions performed by each entity.

Driver Entity

The first entity is the *Driver* entity, which must be defined for every PARSEC program. The entity sets up the simulation, reads the command line and instantiated the various entities. This entity performs a job like the **main()** function in a standard C program. Execution of a PARSEC program is initiated by executing the first statement in the

body of *Driver* entity. The driver entity takes the same **argc** and **argv** parameters as the C main function. Parameters recognized by the PARSEC runtime system is removed from **argc** and **argv** before the driver is invoked. In this simulation, the entities are created by using the **new()** statement. For example, the four new instances of the various entities created by *Driver* entity are as follows,

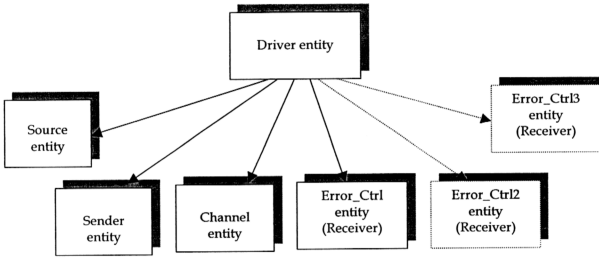

Figure 4.1: The new instances of the various entities created by Driver entity

The five entities (in Figure 4.1) which make up the simulation interact via message passing. Data is passed from one entity to another entity. The interface between the *Source* and *Sender* entity depends on the source type. As mentioned previously, the source types implemented are *speech* and *data* sources. When a *speech* source type is used, data is generated and sent 32 kbps. When a *data* source type is used, the *Sender* uses *DataReqMsg* message to initiate the transfer of data from *Source* to *Sender* entity and also sent at 32 kbps. The 32kbps were chosen as a reasonable rate for encoded speech and data. It could be faster or slower, depending on the type of encoding. The remaining interfaces are constant. Data is sent from *Source* to *Channel* in the same way acknowledgements are sent from *Receiver* to *Channel*. Either entity must request the

channel's attention with request to send message, and wait for the clear to send response before sending the data.

The entities defined for the simulation and their interaction with each other are shown in Figure 4.2 and 4.3.

Source Entity

The entity pulls data from an input file and sends it to the *Sender* entity. The input file contains set of data created by the *Source* entity. The *Source* entity is responsible for generating traffic based on exponentially distributed On/Off period. Source entity is created by the driver entity across simulation time as a Poisson process (calls/second). Source has to consult with *Sender* entity which buffer is stored before they start generating and sending the traffic. In another word, this entity is primarily responsible for generating workload to be used on transmission.

The entity communicates with the *Sender* entity by sending a *NewDataMsg* message for speech source type. For data source type, the entity sends that message after receiving *DataReqMsg* from the *Sender* entity. The difference is caused by feeding style either on demand as would be the case in a file transfer, or at regular intervals, as if a source of multimedia data.

Sender Entity

This is the second entity in the simulation that transmits data to the *Receiver* entity. The *Sender* entity must then segment the data and add error control coding as the data arrives. The segmentation process will be explained more details in the Section 4.2.4.

After finish performs this process, the entity then sends a request message, *ReqToSendMsg*, to the *Channel* entity. If the access request is granted it sends a *ClearToSendMsg* message to this entity. After this notification, the *Sender* entity sends a *PacketMsg* message which contains a packet to the *Channel* entity. If acknowledgement is activated, it waits for an acknowledgement before proceeding. If after a timeout, no acknowledgement arrives, a retransmission is performed. Error control coding such as is applied here as required.

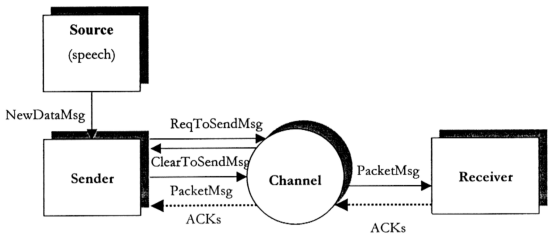

Figure 4.2: Entities and messages passed between them *(speech source type)*

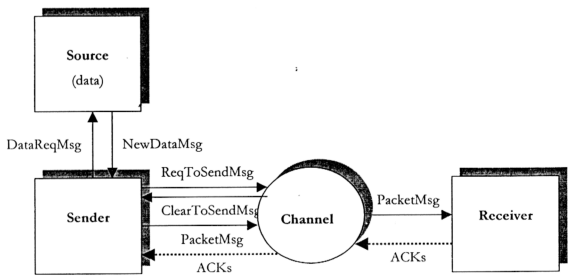

Figure 4.3: Entities and messages passed between them *(data source type)*

***Channel* Entity**

The *Channel* entity waits for requests to send a message from another entities and grants access if the channel is not in use. If access granted, no further request is serviced until a packet has been taken in. Errors are added to the data and will then be attached to the *PacketMsg* message before they are passed to the receiver's queue with a delay factor specified by the channel delay. If, based on the channel model, a bit is deemed to be subject to error, it is simply flipped (changed from 1 to 0 or vice versa). The channel model that is used to make this decision is two-state DTMC. The mechanism on how to do this was explained in Section 3.3. Any time a *Sender* and *Receiver* entity want to communicate with the *Channel* entity, it must assert a *ReqToSendMsg* message which the channel responds to with a *ClearToSendMsg* message if it is free, and then waits for a packet only from the Sender entity who was granted access.

***Receiver* Entity**

The *Receiver* entity is the final destination for a packet after parsing through the *Channel* entity. This entity waits for packets to arrive from the *Channel* entity via *PacketMsg* message. If acknowledgement is enabled and the packet is error free, then an acknowledgement is sent to the *Channel* entity. Otherwise, no further action is taken. Other messages involved in communication between both entities are *ReqToSendMsg* and *ClearToSendMsg*.

4.2.2 System Parameters

The eight system parameters, listed in Table 4.1, were varied to allow us to examine the adaptive error control under various channel conditions.

Parameter	Possible Values
ARQ type	none, Stop-and-Wait, Go Back N, CACK, SACK
Round Trip Time	number of time units to allow a packet to be outstandings
Error Control Coding	none, checksum, Reed Solomon, Viterbi Code
Packet type	ATM packet
Max allowed delay	number of simulation clock ticks a packet is allowed to remain buffered
Source Type	Data, speech
Channel Model	pedestrian, car speeds
Bit Error Rates	Good, Bad states

Table 4.1: System Parameters

The system parameter and their possible values are elaborated as follows.

ARQ Type

The ARQ type may be set to one of five values as shown. The five values are none, Stop-and-Wait, Go-Back-N and Cumulative Acknowledgement (CACK) and Selective Acknowledgement (SACK). If no acknowledgment is used, data flow is unidirectional through the channel. The sender entity generates data and places it in the channel entity as quickly as it can. It is constrained only by the data rate of the channel entity and of the source entity until it has no more data to send. The receiver entity accepts the data at this rate and stores it to an output file, possibly after decoding for error correction. Alternatively, the raw data may be stored to a file directly if no error correction coding was requested. In any case, the sender has no knowledge of the outcome of the transmission.

The available ARQ schemes are essentially those described in the literature (Lin, 1983) and will not be described in detail here. When one of these ARQ schemes is used, the simulation limits the send buffer to eight packets, and in the case of SACK, the receive buffer to the same amount. In all cases, the acknowledgment packet consists of four bytes. The first two bytes are a sequence number related to the packet being ACKed, while the last byte is a checksum to guard against error. The third byte is used only for the SACK scheme, where it is necessary to provide the sender with information about any packets which have arrived early. Each bit of this byte corresponds to one of these packets, referenced starting with the last one received. So, this format is use to prevent the packet loss and to organize the entire packet in both sender and receiver entity. In the case of packet loss occurred, then the acknowledgement packet will send the information to Sender to retransmit the loss packet again.

Error Control Coding

The choice of error control coding is another factor which affect the results in the simulation. As is shown in Table 4.1, there are four options. If no coding is used, the data is simply delivered to a file regardless of bit errors. Alternately, we may employ a simple error detection scheme to determine packets which are in error. Taking a step further, we may use an error correcting code such as a block code or a convolutional code. For the block code, we chose a Reed-Solomon whose symbol size and block length vary depending on the packet size. The convolutional code is decoded with a rate ½ hard decision Viterbi decoder and the outcome of this 'maximum likelihood' decision is verified with a checksum. This particular code is independent of the packet type since it codes a continuous stream of data.

In all cases, the packet header contains an additional checksum whose failure results in a lost packet irrespective of the outcome of the other codes. It is assumed that if a packet header is not reliable, nothing can be done with the data. At the very least, the packet number contained in the header is unreliable, and most of the ARQ schemes rely on this information to effect their algorithms.

Cyclic Redundancy Code

The Cyclic Redundancy Code (CRC) is a very powerful but easily implemented technique to obtain data reliability. The CRC technique is used to protect blocks of data called Frames. Using this technique, the transmitter appends an extra n-bit sequence to every frame called Frame Check Sequence (FCS). The FCS holds redundant information about the frame that helps the transmitter detect errors in the frame. The CRC is one of the most used techniques for error detection in data communications.

Retransmission Policy

The retransmission policy in the experiment is to send the oldest packet not yet acknowledged or known to be in transit. When a packet is sent, it is marked in transit until it is either acknowledged, at which time it is replaced in the buffer by new data, or until it times out, as per the Round Trip Time (RTT). Each packet is effectively checked continuously for the expiration of its RTT, after which time it is marked for retransmission.

Packet Size

This option allows for the segmentation of source data into arbitrarily sized blocks. In any case, the data fed from the source is fragmented according to the information size provided when using this switch. The packet size dictates the largest atomic unit which is to have error control coding applied to it, and which is to then pass through the channel.

When encoding with Reed-Solomon, this option is also used to specify the parameters of the block coding, which is necessarily related to frame size. In particular, the code rate, as described in the previous section, is fixed by naming the values for the symbol size, block size, and amount of redundancy data, along with the frame size.

Delay Time

For real time applications, a packet which has exceeded a certain specified age limit is no longer useful to the application. Therefore, the user may specify that packets which have aged beyond a certain point, as per the maximum allowed delay, be dropped at the sender. Specifying zero for this parameter leaves the delay unbounded which may be useful to for example a file transfer operation, where delay is not critical.

4.2.3 Experiment Design

In this section, we illustrate more details about entire simulator program. The first job of the simulation is to read data from an input file and then passing it to the sender entity. An input file is provided for various error control strategies and others parameter. The content of the input file has been explained in the previous section (See Section 4.2.1). This input file can be read by a driver entity. In order to model different

kinds of sources, the input file contains two type of data source; a speech and data type source. For data source type, the *Sender* entity is required to request for the data first and then followed by feeding from the *Source* entity. For speech source type, the data was fed directly to the sender without requesting it first. The difference is because we want to know the performance of both these source types in term of normal and multimedia data under given channel condition. This situation is depicted in Figure 4.4.

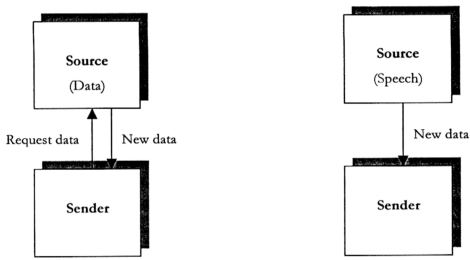

Figure 4.4: The difference between Data and Speech transmission

Next, the sender must segment the data and form the packet. In our experiment, we only test ATM packet,type. Data is simply broken into chunks of a certain size (the segment size). This option allows for the segmentation of source data into arbitrarily sized blocks. In any case, the data fed from the source is fragmented according to the information size provided when using this switch. The packet size dictates the largest atomic unit which is to have error control coding applied to it, and which is to then pass through the channel. The *Sender* entity then requests for a *Channel* entity and sends a packet when the *Channel* is granted. If acknowledgement is activated, the

Sender entity must wait for an acknowledgement before proceeding. If after a timeout, no acknowledgement has arrived, a retransmission is performed. The *Sender* entity will also automatically remove packets which no longer meet the delay constraint. The following source code shows how to determine the packet time out.

```
if ((status[i]==SENT) && (now-sent_time[i] > rtt)) {
        status[i] = FILLED; }
```

Before the *Sender* entity passes the packet to the *Channel* entity, it must assert a *ReqToSendMsg* message which the channel respond to with a *ClearToSendMsg* message if it is not in use, and then waits for a packet only from the sender who was granted access. At the same time, no further request is serviced until a packet has been taken in. After receiving the packets from the sender, the channel then performs bit errors injection to the data according to the channel model used in this simulation. It also means, the simulator examines each packet one bit at a time. On each bit, it decides whether to apply an error with a probability that depends on the current state, and further decides whether or not to change state, according to the transition probabilities. The command below shows the message passed between both the sender and channel entities. It also shows that, the channel is only ready to receive a new packet if it is in free.

```
receive (ReqToSendMsg rts) {
        send ClearToSendMsg to rts.sender;
            receive ( PacketMsg pkt) {
```

After a suitable delay corresponding to the delay incurred in the radio hardware, in our simulations we fixed at 0.5 ms, each packet is then passed to the receiver's queue where it is decoded and handed to the sink which may or may not generate transport

layer acknowledgments. In this case, if packet has a delay more than 0.5 ms then it will assumed to be drops. Based on Lettieri (1997), a delay of more than 0.5ms would result in a two-way interactive speech becoming blurred. The following command shows how the packet was transmitted from channel to the destination, but ensures that the receiver does not 'see' the packet until after the channel delay.

```
send PacketMsg {self, temp.rxid, temp.txid, temp.data, err, temp.size, temp.gen_time} to
temp.rxid after channel_delay;
```

Upon receiving a packet from the channel, the decoding process is performed on the packet. If any error is detected after error checking is performed, a suitable error control coding is applied. The error can be detected from the *PacketMsg* message which is received from the channel entity. The error control coding options that may be applied are Checksum, Viterbi code and Reed-Solomon code. Otherwise, we assume that it is a good packet. The receiver is also required to request for a channel before sending an acknowledgement through the channel back to the sender. However, the type of acknowledgment depends on whether ARQ was called for at the beginning of the session.

```
send ReqToSendMsg {self} to temp.channel;
    receive (ClearToSendMsg cts) {
```

If there is no more power to transmits data to the destination, then the simulation terminates and statistics are produced for the user. In another word, the selection of error control scheme and the amount of data to be sent are depending on current available power. The among information generated are energy consumed by both

sender and receiver, maximum and average delay seen by the packets, amount of data dropped due to excessive delay and residual BER in the output data.

Table 4.1 lists a sample of the more important user selectable parameters and their possible values. This way, statistics are collected for different error control schemes, including different FEC and ARQ types as well as all the possible hybrids for a variety of source types, frame sizes, and so on.

4.3 Assumptions

It is necessary for us to make several assumptions about the simulation environment before we get the result of the simulation just described. First, since the channel model explained previously is used to represent the wireless link, the selection of a number of parameters is required. In particular, we choose to fix the Bad and Good states BER at 0.5 and 0.00001 respectively. Bad state BER = 0.5 simply means that when the channel is in the bad state, we have a 50-50 chance of a bit being in error. So for each bit that goes by in this state, half of them are flipped. Good state BER of 0.00001, means that in the good state, 1 out of 10,000 bits will be in error. The data rate of the channel is taken as 625 kbps for all simulations. This data rate was chosen because it was based on a real world popular wireless LAN card available (Eckhardt, 1998). Furthermore, in this experiment we just focus on small area of wireless network and this data rate is sufficient to simulate our algorithms in wireless LAN.

Second, a perfect CSMA/CA MAC layer is assumed with the sender and the receiver the only nodes on the link. In other words, the sender and receiver have access to the

channel at will and compete only with each other for that channel. Furthermore, if one unit has the channel, the other will wait until the first is done before attempting to transmit. Accordingly, it is assumed that the radio is put to sleep when neither receiving nor transmitting, and that the switch from any one mode to another requires 4 ms. Lastly, the user may specify between two different sets of transition probabilities. In particular, the user may specify one of either of the channel presented in Chapter 3: that seen from a moving car, or that seen at pedestrian speeds.

4.4 Compilation and Execution Environment

In this experiment, we use Windows version of PARSEC compiler which is now available for use with Microsoft Visual C++. The compilation process is not much different from standard compilation of C programs since this simulator is written merely in C language.

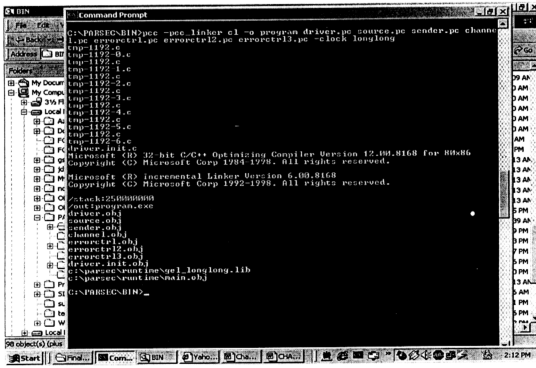

Figure 4.5: Compilation process and generation of executable file

The compilation of a PARSEC program consists of two phase process. In the first phase, the codes are compiled for all its PARSEC constructs. If this phase is successfully compiled without any syntax error, the compilation enters the second phase, which is the compilation of the C language constructs. Upon completion of the second phase, an executable program file is generated by the compiler as shown in the above figure (Figure 4.5).

The various results of the simulation performed are presented in the next chapter, while we save several conclude for the last chapter.

3. Speech transmission at pedestrian speed
4. Speech transmission at car speed

The algorithms were tested under each of these network scenarios for two call arrival pattern, which are normal call arrival pattern and fluctuating call arrival pattern.

5.1.3 Testing Output Parameters

Before we proceed to the results of the tests, there are several output parameters, which are necessary to be produced for each test. The parameters and information are;

a. Number of packets dropped / lost
b. Numbers of packets that arrive at the destination
c. The percentage of packet dropped
d. Simulation time
e. The message passing among the entities

These values are accumulated and printed at the end of the simulation. The duration time for each test depends on the FEC scheme and source of data transmission type they used. An example of simulation system interface is shown in Figure 5.2.

5.1 Testing

5.1.1 Description of the Simulation Scenario

In the development and testing of this simulation implementation, the network scenario below is used.

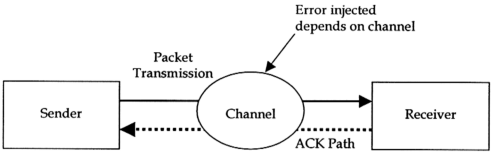

Figure 5.1 Network Scenario for the Simulation

For this project, the simulation was built and compiled using the windows-based version of PARSEC compiler. We used the Microsoft Windows 2000 Professional as the operating system for compiling and executing the simulation.

5.1.2 Types of Test

All the three algorithms proposed in Chapter 3 were evaluated under different traffic environments and algorithm parameter settings. The algorithms was tested for four different network scenarios as follows:

1. Data transmission at pedestrian speed
2. Data transmission at car speed