Analysis of real-time control systems by the model of packet nets

by MOHAMED GAWDAT GOUDA
Honeywell Corporate Technology Center
Minneapolis, Minnesota

INTRODUCTION

During the planning stages of a real-time control system, the planning team is more concerned about the global aspects of the system rather than the details of its different components. The first questions that have to be resolved are, What are the basic processes (or modules, or tasks) in the system? What are the data flow rates between these processes? and What are the memory and processing rate requirements of each process? Also of prime concern is how to select a suitable computer architecture to host the system and how to decompose the system on a selected distributed architecture.

In order to be able to address these questions in a systematic and formal way, we introduce a model, called Packet Nets, for real-time control systems. The model is based on the concepts of data flow, but unlike other data flow models, this model focuses on the global system characteristics and hides the details which are not needed during system planning. The model is intended as a tool to represent real-time control systems in an implementation-independent fashion. Moreover, it can be used in estimating the values of some basic system parameters such as data flow rates between system components and the processing rate requirement of each system component.

The model in Figure 1 is restrictive as each process can receive data packets from only one process and can send data packets to only one process. To remove these restrictions, we extend the model such that a process can have more than one input and one output queue. As shown in Figure 2, the different input queues of a process' are controlled by an OR-receiving operation or by an AND-receiving operation. Similarly, the different output queues are controlled by an OR-sending or an AND-sending operation. These four sending/receiving operations are defined (informally) as follows.

To execute the OR-receiving operation, the process waits until one of its input queues has at least one packet. Then, the process removes one packet from this input queue and proceeds to process it. To execute the AND-receiving operation, the process waits until there is at least one packet in each input queue. Then, the process removes one packet from each input queue and proceeds to process them.

To execute the OR-sending operation, exactly one of the associated output queues is selected (arbitrarily) to put the sent data packet in it. To execute the AND-sending operation, one data packet is put in each output queue. The sent packets are not necessarily identical.

A process with an OR-receiving operation and an AND-
sending operation is called OR-AND process. Similarly, OR-OR, AND-AND and AND-OR processes can be defined. A short-hand notation for the different classes of processes in a packet net is shown in Figure 3. Observe that if a process has a single input queue, then an OR-receiving operation becomes equivalent to an AND-receiving operation. Therefore, we use the convention that if a process has a single input (or a single output) queue, then it has an OR-receiving (or OR-sending respectively) operation.

To give an example of a real-time control system and its packet net, consider a radar scheduling system. The system receives requests to use the radar. It orders these requests based on some priority scheme, then delivers the ordered sequence (called frame) of requests to the radar to serve them in order. But since the radar cannot accept any frame unless it satisfies some constraints, the system has also to examine the generated frame against all the constraints (imposed by the radar) before delivering the frame to the radar.

Figure 4a shows the outline of a radar scheduling system. Arriving requests are of three different classes. Each request has its own local priority such that requests of the same class are ordered according to their local priorities. The partially sorted requests are sent to the framing process to be ordered in a time frame based on some global priority. The frame is then sent via a frame modification process to a number of checking processes. Each checking process examines (in parallel with other checking processes) whether or not the frame meets some radar constraint. The individual decisions for the frame are then sent to a final decision process to decide whether the frame is accepted or rejected. Accepted frames are sent to the radar, while rejected frames are sent back to the frame modification process to be corrected before being examined once more. The packet net for this system is shown in Figure 4b.
PART OF THE ANALYSIS OF A REAL-TIME SYSTEM IS TO COMPUTE THE RATES WITH WHICH THE DATA FLOW BETWEEN THE DIFFERENT SYSTEM COMPONENTS. IN THE PACKET NET MODEL, THIS CORRESPONDS TO COMPUTE THE PACKET FLOW RATE IN EACH QUEUE IN THE NET. TO MAKE THIS COMPUTATION, THE FLOW RATE (IN PACKETS/SEC.) IN EACH EXTERNAL INPUT QUEUE SHOULD BE ESTIMATED FROM THE APPLICATION (E.G., THE RATE WITH WHICH THE SENSOR DATA ARRIVE TO THE SYSTEM). IF AN INPUT FLOW RATE CHANGES WITH TIME, AN UPPER BOUND FOR ITS VALUE SHOULD BE ESTIMATED. ALSO, THE PROBABILITIES WITH WHICH EACH OR-SENDING OPERATION DISTRIBUTES THE OUTPUT PACKETS AMONG ITS OUTPUT QUEUES SHOULD BE ESTIMATED FROM THE APPLICATION.

AS AN EXAMPLE, CONSIDER THE PACKET NET IN FIGURE 5. THE EXTERNAL INPUT FLOW RATE TO THE NET IS ESTIMATED TO BE 10 PACKETS/SECOND. THE OUTPUT QUEUES OF OR-OR PROCESS 1 ARE LABELLED WITH THE PROBABILITIES .2, .1 AND .7, WHILE THE OUTPUT QUEUES OF OR-OR PROCESS 2 ARE LABELLED WITH THE PROBABILITIES .5 AND .5. THE VARIABLES $T$, $U$, . . . , AND $Z$ ARE ASSOCIATED WITH THE DIFFERENT QUEUES IN THE NET; EACH VARIABLE IS DEFINED TO BE THE PACKET FLOW RATE IN ITS ASSOCIATED QUEUE. THE PROBLEM, NOW, IS TO EVALUATE THESE VARIABLES. NEXT, WE PRESENT THE EQUATIONS TO EVALUATE THE PACKET FLOW RATES IN ANY WELL-FORMED PACKET NET.

CONSIDER THE OR-OR PROCESS SHOWN IN FIGURE 6A. THE INPUT PACKET FLOW RATE TO THAT PROCESS EQUALS $\sum_{j=1}^{m} x_j$. THIS FLOW IS DISTRIBUTED AMONG THE $n$ OUTPUT QUEUES according to their associated probabilities. Thus, the packet flow rate $y_i$ in the $i$th output queue is evaluated as follows:

$$y_i = p_i \sum_{j=1}^{m} x_j$$

$i = 1, \ldots, n$

Similarly, the packet flow rate $y_i$ in the $i$th output queue of an OR-AND process (Figure 6b) is evaluated as follows:

$$y_i = \sum_{j=1}^{m} x_j$$

$i = 1, \ldots, n$

Consider the AND-OR process shown in Figure 6c. Be-
cause of the AND-receiving operation in this process, the flow rates in all input queues should be equal. Assume that the flow rate in any input queue equals $x$. This flow is distributed among the $n$ output queues according to their associated probabilities. Thus, the packet flow rate $y_i$ in the $i$th output queue is evaluated as follows:

$$y_i = p_i x \quad i = 1, \ldots, n$$

Similarly, the packet flow rate $y_i$ in the $i$th output queue of an AND-AND process (Figure 6d) is evaluated as follows:

$$y_i = x \quad i = 1, \ldots, n$$

For any packet net, we can write the flow equations using the general forms just discussed. By solving these equations simultaneously, the packet flow rates in each queue in the net can be determined. For example, the flow rate equations for the net in Figure 5 can be written as follows:

\begin{align*}
t &= .5 u \\
u &= .2 (t + 10) \\
v &= .1 (t + 10) \\
w &= .7 (t + 10) \\
x &= .5u \\
y &= w \\
z &= x = v
\end{align*}

Thus, the flow rates can be determined: $t = v = x = z = 1.11$, $u = 2.22$, $w = y = 7.78$. All dimensions are in packets/second. If the packet sizes (in bits/packet) for each queue can be estimated, then the flow rates can be computed in bits/seconds.

For many packet nets, the flow equations are inconsistent; thus produce no solution. For example, if the output probabilities of Process 2 in the packet net in Figure 5 are...
changed from .5 and .5 to .4 and .6, we get the following flow equations:

\[
\begin{align*}
t &= .4u \\
u &= .2(t+10) \\
v &= .1(t+10) \\
w &= .7(t+10) \\
x &= .6u = .12(t+10) \\
y &= w \\
z &= x
\end{align*}
\]

From the third and fifth equations, we have

\[v \neq x\]

This contradicts the requirement that \(v\) and \(x\) should be equal since they are AND inputs to Process 4 in the net.

A packet net is said to be well formed only if its flow equations are consistent and produce a single solution to the flow rates in the net. Obviously, the packet net in Figure 5 is well formed; but if we change the output probabilities of Process 2, the resulting net is not well formed.

The consistency of flow equations for a packet net depends on the net structure, and the estimated output probabilities for the processes in the net. Therefore, inconsistent flow equations imply that either the net "structure" is intrinsically inconsistent, or that our estimation of the output probabilities is inconsistent. In the previously-mentioned example, we illustrated how an inconsistent estimation of the output probabilities can lead to an inconsistent set of flow equations. Next, we give some examples of intrinsically inconsistent packet nets.

Figure 7a shows a part of a packet which consists of a cycle with \(n\) OR-AND processes. The flow equations for such cycle are as follows:

\[
\begin{align*}
x_1 &= x_n + I \\
x_2 &= x_1 \\
\vdots \\
x_n &= x_{n-1}
\end{align*}
\]

From these equations, we get \(x_n = x_n + I\) (which can be true iff \(I=0\)). Thus, in Case \(I\neq 0\), this cycle is intrinsically inconsistent.

As another example, consider a cycle of \(n\) AND-OR processes (Figure 7b). The flow equations for this cycle are as follows:

\[
\begin{align*}
x_n &= I \\
x_1 &= p_1x_n \\
x_2 &= p_2p_1x_n \\
\vdots \\
x_n &= p_np_{n-1}\ldots p_1x_n
\end{align*}
\]

which can be satisfied iff \(p_1 = p_2 = \ldots = p_n = 1\); i.e., each process should have exactly one output queue. If any process has more than one output queue, then the cycle is intrinsically inconsistent.

From these examples it is clear that a packet net should not contain a cycle whose nodes are all OR-AND processes.
or whose nodes are all AND-OR processes. It is also clear that flow equations are useful in designing well formed packet nets.

EXECUTION RATES

Another part of the analysis of a real-time system is to compute the execution rates (in inst./sec.) for all the processes in the system. In order to make this computation, the number of instructions in the main cycle of each process (see Figures 1a and 2) in the net should be estimated.

Assume that the input packet flow rate to a process $P$ is $\sum x_i \cdot x_i$ packets/second. Therefore, the cycle of $P$ should be executed $\sum x_i \cdot x_i$ times per second. If the cycle of process $P$ has $r$ instructions, then the required execution rate of $P$ is $r \cdot \sum x_i \cdot x_i$ inst./sec. Observe that the above computation does not depend on the process type (e.g. OR-OR, OR-AND, ...).

The required execution rate of a process can be reduced by adding extra copies of the same process to the packet net. For instance, Figure 8a shows an arrangement of two processes $P$ and $Q$ connected by a queue (from $P$ to $Q$). Assume that the input packet flow rates to $P$ and $Q$ are $x$ and $y$ respectively. Thus, the required execution rates for $P$ and $Q$ are $ux$ and $vy$, where $u$ and $v$ are the number of instructions in the main cycles of $P$ and $Q$ respectively. Assume that an extra copy $Q'$ of process $Q$ is added to this arrangement, as shown in Figure 8b. This requires that process $P$ becomes responsible for equally distributing its output packets to both $Q$ and $Q'$. In other words, $P$ becomes an OR-OR process with output probabilities of $.5$ and $.5$. The input packet flow rate to $Q$ (or to $Q'$) becomes $y/2$; and the execution rate of $Q$ (or $Q'$) becomes $vy/2$ half the original amount. On the other hand, process $P$ has more responsibilities; and the number of instructions in its cycle increases with the amount of $\delta$. Hence, its execution rate will increase to become $(u+8)x$ inst./sec.

In the previous example, the distribution of data packets between the process copies is handled by an already existing process in the net. In some cases, however, a new process is added to the net to perform this distribution function. We call such a process a distributor. Figure 9 shows how $(k-1)$ additional copies of one process and a distributor are added to a packet net to reduce the process execution rate by a factor of $1/k$. Observe that the distributor execution rate $\delta y$ is “small” since there is a “small” number of instructions (namely $\delta$) in its cycle.

Another advantage of adding extra copies of the same process to a packet net is to increase the overall system reliability and availability. At any rate, whether the extra process copies are added to decrease the execution rate requirements or to increase the system reliability, the implied assumption is that no two (or more) copies of the same process should be assigned to the same processor in the host hardware. The problem of assigning processes to processors is discussed in more detail later on in the sixth section; but first we need to discuss the memory requirements of packet nets.

MEMORY REQUIREMENTS

Memory is required for a packet net to store its processes and its queues. The required memory to host one process...
in the net depends on the control and data structure of that process; i.e., it depends on the size of its program and its data file(s), if any. An upper bound for the required memory of each process should be estimated from the application so that the total memory requirements can be calculated.

The required memory to host the queues of a packet net depends primarily on how this memory is organized and how it is managed. We next describe one scheme for the organization and management of such a memory. Other attractive schemes are not discussed in this paper because of the space limitation. For convenience, we define an OR process to be either an OR-OR process or an OR-AND process, i.e., it is a process with an OR-receiving operation. Also, we define an AND process to be either an AND-OR process or an AND-AND process.

Consider an OR process $P$ with $m$ inputs (Figure 10a). To allow both $P$ and its input processes to proceed concurrently, the input queues of $P$ should be implemented as one packet pool which can hold up to $m+1$ packets (Figure 10a). When one input process (say, $Q$) finishes the assembly of a new packet (say, $p_1$) in the packet pool, and $P$ finishes the processing of an old packet (say, $p_2$) from the packet pool, then $P$ initiates a packet interchange with $Q$. Thus, $Q$ starts to assemble a new packet in place of $p_2$, while $P$ starts to process the packet $p_1$.

Similarly, if $P$ is an AND process with $m$ inputs, then the packet pool between $P$ and its input processes should hold up to $2m$ packets (Figure 10b). When each input process finishes the assembly of a new packet, and $P$ finishes the processing of the $m$ old packets, then $P$ initiates a packet interchange which causes $P$ to get the $m$ new packets and assigns each input process a space for one packet.

In this static memory allocation scheme, the storage capacity of any cycle in the packet net is fixed. Thus, it is possible that packets may continue to flow into some cycle in the net until the cycle is completely full. At this instant, the cycle processing stops and a deadlock situation arises. The remainder of this section examines such deadlocks in more detail. In particular, we introduce some sufficient conditions for deadlock avoidance.
Deadlocks arise in a packet net whenever one or more cycles in the net become full of packets. But, because AND processes can never increase the number of packets in any cycle (as illustrated in Figure 11), they can never contribute to deadlock situations. Therefore, we need only to consider OR processes. In general, if a cycle contains one or more OR processes in a packet net, the OR processes can fill the cycle with packets causing a deadlock. In order to avoid such a deadlock, we need sufficient conditions to prevent the OR process from filling their cycles with packets. Next we present such a set of conditions; but first we need to define "cycle queues." A cycle queue is a queue which exists in one or more cycles in the packet net.

The following three conditions on the structure and operation of OR processes are sufficient to avoid deadlocks in packet nets:

**cond1**—Each OR process has at most one input cycle queue (Figure 12a). A packet which arrives to the OR-process via the input cycle queue is called a cycle packet; otherwise it is a non-cycle packet.

**cond2**—If there are one cycle packet \( p_i \) and one space \( p_j \) for a new packet in the packet pool of an OR process \( P \), then \( P \) initiates a packet interchange such that \( p_j \) is assigned to the input cycle queue to assemble a new cycle packet, and \( p_i \) is assigned to \( P \) to be processed. However, the processing of \( p_i \) will not start until there is one packet space in each output queue of \( P \).

**cond3**—If there are one non-cycle packet \( p_i \) and one space \( p_j \) for a new packet in the packet pool of an OR-process \( P \), and if there is one packet space in each output of \( P \), then \( P \) initiates a packet interchange such that \( p_j \) is assigned to the input non-cycle queue (via which \( p_i \) has arrived), and \( p_i \) is assigned to \( P \) for processing. The processing of \( p_i \) can start immediately after the packet interchange.

Observe that both **cond2** and **cond3** do not restrict the structure of packet nets in any way; they merely enforce some order on the packet processing by the OR processes. On the other hand, although **cond1** does restrict the structure of packet nets, we feel that the restriction is not severe; the packet net in Figure 4b satisfies **cond1**.

The processing of cycle packets can never increase the number of packets in any cycle in the net; thus, it can never lead to a deadlock. For this reason, **cond2** implies that the processing of cycle packets should proceed whenever possible (Figure 12b).

The processing of one non-cycle packet by an OR process can generate at most one extra packet in every cycle which contains the OR process. Therefore, the third condition **cond3** implies that the processing of one non-cycle packet...
shouldn’t start unless the OR process “sees” two available packet spaces in each cycle which contains the OR process. This is the case when the OR process has one packet space in its input pool, and has one packet space in any of its output pools. Under this condition, the processing of non-cycle packets can never make any cycle full of packets.

Because of the space limitation, the discussion in this section has been limited to static memory allocation where each queue has been assigned a fixed memory size. We acknowledge, however, that static memory allocation can be unaffordable in some applications. The solution for these applications lies in dynamic memory allocation where queues can grow and shrink in size as their needs change with time.

DISTRIBUTED COMPUTER SYSTEMS FOR REAL-TIME CONTROL

In this section, we investigate some of the problems associated with designing distributed computer systems for real-time control. Our bias toward distributed computer systems stems from our conviction that these systems provide high degrees of extensibility, integrity and performance which are most needed in real-time environments.

In the previous sections, we have discussed how to calculate the following quantities:

1. The data flow rates (in packets/second, or equivalently in bits/second) between the net processes.
2. The execution rate (in instructions/second) of each process in the packet net.
3. The memory requirements (in words) of each process in the net.

These quantities are needed for the design of a distributed computer system to host a packet net (i.e., a real-time application). For that reason, these quantities are recorded on a graph, called the system graph of the packet net. The nodes in the system graph correspond to bi-directional communication between processes in the packet net. As an example, Figure 13b shows the system graph for the packet net in Figure 13a.

Each node \( i \) in the system graph is labelled with two quantities \((e_i, m_i)\), where \( e_i \) (in instruction/second) is the execution rate of the corresponding process, while \( m_i \) (in words) is the memory requirement for the corresponding process.
process and its input buffer. Each edge \((i, j)\) in the system graph is labelled with a quantity \(c_{i,j}\) (in bits/second) which is the sum of the data flow rate from process \(i\) to process \(j\) and the data flow rate from process \(j\) to process \(i\).

The system graph of a packet net contains all the needed information to design an "optimum" distributed computer system to host the packet net. In Reference 7 we outline a design methodology of distributed computer systems based on the model of system graphs. In this methodology, a system graph is used to generate a set of possible distributed architectures which can host the packet net. Then, from these generated architectures, an "optimum" architecture is chosen based on cost and reliability considerations. This approach still needs more research and more design automation support before it can be utilized in a practical way.

A more practical approach is to begin with a ready-made distributed computer system which can host the packet net. Then, find the "optimum" assignment of the packet net processes to the system processors. This optimization problem can be expressed and solved in terms of system graphs as illustrated by the following example.

Consider the shared bus system shown in Figure 14. It consists of some processors; each of them has its own private memory and its own Bus Interface Unit (BIU). The processors communicate only by sending and receiving messages via the global bus which they control in a decentralized fashion. In this architecture, the global bus is the limiting resource. Thus, on assigning processes to processors in this architecture, the object should be to minimize the bus traffic.

Assume that a shared bus system with \(r\) processors is to host a packet net. The optimum process assignment can be reached by finding at most \(r-1\) minimum cuts for the system graph of the packet net. These cuts should satisfy the following two conditions:

1. \(\Sigma e_i \leq E\) partition
2. \(\Sigma m_i \leq M\) partition

where \(E\) is the processing rate which can be devoted to the application by one processor in the system, and \(M\) is the memory which can be devoted to the application by one processor in the system.

Finding these minimum cuts may require, in general, an exhaustive search among all possible cuts.

It has been mentioned in the fourth section that for reliability reasons, different copies of the same process should be assigned to different processors. To make sure that the minimum cuts technique will produce this assignment, we add an edge to the system graph between any two nodes which correspond to copies of the same process. These edges will be labelled with \(-\infty\), as in Figure 15; thus, the minimum cuts must cut through all of them.

Sometimes it is required (e.g., for input/output considerations) to assign some processes to specific processors. To make sure that the minimum cuts technique will produce these assignments, the following additions should be made to the system graph:

1. A node labelled \((0,0)\) should be added for each processor which needs to host specific process(es).
2. An edge labeled \(-\infty\) should be added between any two processor nodes.
3. An edge labelled \(\infty\) should be added between a processor node and a process node if it is required to assign the process to the processor.

An example is shown in Figure 16.
CONCLUSIONS

A simple model to represent a wide class of real-time control systems is introduced. The model can be used in the early stages of system planning to represent the system in a gross form while hiding the fine details till later stages. The model can be also used to compute the values of different parameters which are critical to the planning of real-time systems. The required computations are simple and can be constructed and executed in a systematic way. The model is also useful in planning the computer system to host the application.

It is conceivable that for specific applications, more features (or restrictions) can be added to (or imposed on) the model to "tune" it to that particular application. In this presentation, however, we have only discussed those features that seem common to a wide class of applications.

ACKNOWLEDGMENTS

Some of the ideas in this paper have been motivated by discussions with R. G. Arnold and E. D. Jensen. I am also thankful to the referees for their helpful comments.

REFERENCES
