Communication Primitives for Real-Time Distributed Synchronization over Small Area Networks

Sang-Hun Lee and Hyun-Wook Jin
Department of Computer Science and Engineering, Konkuk University
Seoul 143-701, Korea
{mir1004, jinh}@konkuk.ac.kr

Abstract—Many emerging embedded systems consist of multiple embedded nodes, which are connected through an internal small area network. In such systems, the synchronization between the embedded nodes is essential to provide real-time cooperation environment. On small area networks, the severe obstacle for achieving real-time synchronization is the unpredictable overhead of the system software running on the embedded nodes. This is true especially if a traditional operating system has been installed. Despite these drawbacks, it is quite common to run traditional operating systems on some of the embedded nodes to utilize various existing applications. In this paper, we propose a novel design of communication primitives called RTDiP-Sync for real-time distributed synchronization over small area networks. Experiment results show that the suggested design can minimize the communication overhead and provide better overhead prediction.

I. INTRODUCTION

Many emerging embedded systems for machine control perform quite complicated tasks and run various applications. Due to the significant requirements for complicated applications, the embedded systems that equip with few to many embedded controllers or boards have been suggested [1, 2]. In most cases, these multiple embedded controllers are connected through a relatively small area network and collaborate by communicating each other. It is quite common to run traditional operating systems such as Linux and Windows on some of the embedded boards to utilize various existing applications.

In the real-time distributed systems, the synchronization between the nodes is essential to provide real-time cooperation environment. If the network is unreliable and generates large overhead like wide area networks, it drastically disturbs the real-time synchronization. On small area networks, however, the severe obstacle for achieving real-time synchronization is the unpredictable overhead of the system software running on the computing nodes. It is because the network overhead and its uncertainty are very small in case of small area network. This is true especially if a traditional operating system has been installed because they are not designed for real-time and do not provide an efficient way for synchronization. For instance, the communication protocol stacks (i.e., TCP/IP protocol suite) implemented in the traditional operating systems are not designed for real-time data transmission.

In general, the synchronization managers keep interesting information updated through communication between computing nodes. In TCP/IP, the data transmitted to the receiver are passed to the application in sequence without intentional packet drop. Thus, to attain the packet containing the latest information, the synchronization managers have to first drain all the packets that hold outdated information because the latest data for synchronization is queued in the tail. This makes guaranteeing real-time requirements even harder. Moreover, since TCP/IP is implemented inside of operating system, the applications have to use system calls to send or receive data, which induces a lot of additional work such as process scheduling when they return.

To overcome the limitations of legacy communication semantics, new communication semantics for synchronization have been suggested and their potential has been presented [1, 3, 4]. However, its detail design issues and optimal suggestion have not been studied in the literature thoroughly. In this paper, we propose a novel design of communication primitives called RTDiP-Sync for real-time distributed synchronization over small area networks. Especially we target traditional operating system environment because it makes the design issues more complicate. Performance measurements present that the RTDiP-Sync can provide better performance than existing communication protocol stacks with respect to predictability in communication, communication latency, and communication bandwidth.

II. RELATED WORK

In the existing communication semantics provided by TCP/IP, the packets arrived in the kernel buffers are queued until the corresponding application calls the receive system call. The system call delivers the packets queued in the kernel in FIFO manner to the application. In a distributed real-time system, the synchronization manager, for example, can send packets with latest state information periodically to client nodes. The packets are queued on the client node unless the receiver read the packet as soon as it arrives. Therefore, several packets can be accumulated on the client and the most updated information can be visible to the receiver only after all the packets are dequeued. This results in nondeterministic latency and makes guaranteeing the real-time requirements very hard. Though TCP provides the urgent mode, TCP still gives the packets to the application in sequence. This is not suitable for real-time synchronization.

To overcome this limitation, GEM [1], state message [3], and RTDiP [4] have suggested new communication semantics. In the new semantics, the communication
protocols do not queue the packets but keep only the last packet received. Though the previous work has presented the potential of the proposed semantics, the detail design issues for optimal performance have not been studied. In this paper, we deal with the design and implementation issues of such communication semantics in detail.

We are also aware about many researches on the lock–free algorithms for efficient synchronizations between processes [5-8]. Compared with the previous work, our research is more focusing on the design and implementation in viewpoint of system architecture and operating system.

There also have been many researches to provide the real-time communication for user-level processes [9-14]. The suggestions can maximize the benefits of the communication semantics for real-time synchronization.

III. DESIGN ALTERNATIVES

In this section, we suggest three different design alternatives of synchronization communication semantics. Especially we target traditional operating system environment. In the synchronization communication semantics, the communication protocols do not queue the packets but keep only the last packet received. Since the receiver side is trickier to design the new semantics, we mainly focus on the design alternatives of the receiver side.

A. Bottom Half based Design

In this design alternative, we follow the basic design of existing TCP/IP protocol stacks but a thin protocol layer is implemented as a bottom half instead of very heavy protocol layers. In this design, when a packet has been received from the network controller, the interrupt handler simply queues it to the backlog queue. Then the bottom half performs the demultiplexing and keeps the last received packet in the kernel buffer dropping previous ones. Therefore, the bottom half implements the most of functions for the new communication semantics. The system call moves the data in the kernel buffer to the user buffer when the application is ready.

This design can be implemented easily by utilizing the kernel interface to insert a new transport layer. However, it can result in large overheads because when the bottom hal is finishing it checks for other events to be processed and handle them if they are. Also, the bottom half based design harms the overhead predictability. On uniprocessor systems, if the bottom half has a work to do, no applications can run until the bottom half finishes its tasks. Thus, the bottom half can occupy the processor to handle the packets for low priority processes even if a higher priority process is ready to run for urgent deadline.

B. System Call based Design

To overcome the drawbacks of bottom half based design, we can get rid of the bottom half from the data path and efficiently distribute its task into user library, system call, and the network interrupt handler. When the packet is received, the network interrupt handler quickly decides which connection is corresponding to the packet and moves it to the kernel buffer.

The system call is responsible for the data movement between the kernel and user buffers. Since the kernel buffer is shared between the system call and the interrupt handler, we need the synchronization to access the data structure.

The process scheduler assigns the processor to the processes based on their priority. Thus the higher priority process can have more chance to call the receive system call than lower priority processes. The system call based design, however, can still suffer from the overhead fluctuation. This is mainly because the system calls also check for scheduling events and pending signals before returning.

C. Shared Buffer based Design

In this design alternative, we remove both bottom half and system call on the data path. The system call in Section III.B does two important tasks: i) copying data from kernel buffer to user buffer and ii) notifying arrival of new data to user process. This means that we need to take care of these two operations without system call. To achieve this, we suggest utilizing shared buffers between kernel and user.

By sharing the network data buffer between the kernel and the user, we do not need an additional operation (i.e., system call) to copy the data to the user buffer. When the interrupt handler moves the data from network controller to the kernel buffer, the data is also located in the final destination user buffer because the buffers are mapped to the same physical memory page frame(s). We can also use another shared memory area to let the user process know the new packet arrival. The user process can simply notice the new data by polling the shared memory area while the interrupt handler reports the length of data received when it moves the data to the kernel buffer.

However, to implement this design alternative, we have to tackle various tricky issues. First, since the memory mapping is an expensive operation, an efficient way to allocate the shared buffer should be provided. Next, we need to take care of synchronization on shared buffers between kernel and user process otherwise a race condition can happen. Finally, processor cache system must allow both kernel and user process see the same value consistently for the shared memory areas.

IV. RTDiP-Sync: DESIGN AND IMPLEMENTATION

In this section, we detail the novel design of communication primitives for real-time synchronization. We choose the shared buffer based design described in Section III.C. The proposed design can be implemented as a kernel module and user library without any kernel modification. Also it can still support TCP/IP packets without any interference. We extend RTDiP [4] to implement the communication primitives. RTDiP provides a user-level thin transport layer and allows the user process can access the data link layer directly bypassing TCP/IP. We call the communication primitives implemented as RTDiP-Sync.

A. User Library and API

The user library is responsible for sending and receiving user data to/from the RTDiP-Sync transport layer. The APIs
are very simple to utilize at the various applications and middleware. The open and close operations for connection management are done by following functions:

- int RTDiP_Sync_Init (struct ku_sock *k_sock, int port): It allocates internal data structures to manage a connection and passes the reference to them through the ku_sock structure. It also initializes the shared buffers described in Section IV.B.
- void RTDiP_Sync_Close(int port): It releases the resources occupied by the connection.

Actual data transmission is performed by RTDiP_Sync_Send() and RTDiP_Sync_Recv():

- int RTDiP_Sync_Send (struct ku_sock *k_sock, void *data, int length): This function sends a data. Internally it calls a system call (i.e., ioctl) to directly access the data link layer.
- struct RTDiP_Sync RTDiP_Sync_Recv (struct ku_sock *k_sock): This function checks if a new data is arrived in the network buffer. The RTDiP_Sync structure returned includes the pointer to the network buffer for the most recent data, timestamp of the data, and length. If there is no data, the function returns immediately without blocking and the length field has zero value. More details are described in Section IV.C.

B. Management of Shared Buffers

As discussed in Section III.C, sharing the data buffers between the kernel and user can remove both bottom half and system call from the data path. This can prevent the additional overhead generated by entering the kernel and can guarantee the predictable communication overhead.

There can be two different ways to allocate the shared buffers by means of software. One is to simply use an arbitrary buffer in the user memory space. Since the kernel can access the user buffer as it wants the interrupt handler can directly move the data from the network controller to the user buffer. This design, however, works only for a uniprocess system. The reason is that the interrupt generated by new packet arrival can happen at any time. That is, it cannot be guaranteed that the current process is always the corresponding process for the packet arrived. In this case, the interrupt handler can place the data to an invalid user memory area of other process.

The other design is to map the user buffer to the kernel memory space and keep its kernel virtual address instead of user virtual address. Since the kernel virtual memory space is shared between all processes once the buffer is mapped into kernel it can be accessed safely regardless of whether the corresponding process is current or not.

A drawback of this design is the high overhead of memory mapping. In general, the synchronization information is not very large but the memory mapping is done in the page unit, which is usually 4KB. If we apply the memory mapping for large message sizes while removing data copy between user and kernel buffers, the memory mapping overhead can be amortized due to huge benefit of removing the data copy [15]. On the other hand, in case of small messages like synchronization information, the memory mapping rather increases the communication overhead because the benefit of removing data copy is negligible. To resolve this problem, we perform the memory mapping operation only once at the initialization phase (i.e., RTDiP_Sync_Init() described in Section IV.A). In our synchronization communication semantics, only one network buffer is sufficient for each communication connection. Thus we allocate the network buffer when RTDiP_Sync_Init() is called and map it to the kernel memory area. Then the kernel can access the network buffer without mapping overhead during the actual data transmission.

C. Synchronization on Shared Buffers

Since we are sharing the network buffer, the kernel can write on the buffer while the user process reads it (i.e., single-writer, single-reader). This results in race condition. In addition, there should be a mechanism that can notify the user process about the new data – we have decided not to use a system call to do this as discussed in Section III.C. In order to address these issues, we design synchronization between the kernel and the user process based on the algorithms suggested in [6] and [7].

RTDiP-Sync allocates two kinds of shared memory areas as shown in Figure 1: shared buffer and shared info. The shared buffer saves the freshest data as presented in Section IV.B. To tackle the single-writer, single-reader synchronization issue on the shared buffer, we use two shared buffers (shared_buffer0 and shared_buffer1 in Figure 1) like double buffer algorithm [6]. With two shared buffers we can coordinate the kernel and the user process so that they can avoid accessing the same data buffer at the same time.

![Figure 1. Shared Memory Areas: Shared Info and Shared Buffers](image)

This coordination is done through the shared info that includes the shared buffer id (bufferID in Figure 1) on which the kernel accesses and the length of data in each shared buffer (len0 and len1 in Figure 1). When a new data is arrived, the kernel moves it into the shared buffer specified by the shared buffer id. Then the kernel sets the data length information of the corresponding shared buffer. The user process obtains the reference (i.e., pointer) to the shared buffer through RTDiP_Sync_Recv() if the length is larger than zero. If this is the case, the RTDiP_Sync_Recv() function switches the buffer id and resets the length of the shared buffer to zero internally. To avoid a race condition, switching the buffer must to perform in atomic. In general, the atomic operation is implemented...
by enabling and disabling interrupts. This, however, requires entering to the privileged mode. Since we are trying to avoid entering to the kernel, a better way is to use an atomic user-level instruction provided by the processor. For instance, ARM processors support the SWP instruction that swaps the contents of memory with the contents of register in atomic.

D. Cache Consistency

As we have described in Section IV.C, since the kernel and the user process share the same memory area, its contents have to be consistent across user and kernel. The cache can be located between the processor and the Memory Management Unit (MMU) (i.e., logical cache case), or between the MMU and physical memory (i.e., physical cache case). The logical cache stores data with virtual address while the physical cache stores memory using physical addresses. For examples, the ARM processor family such as Intel XScale and ARM7 through ARM10 use logical cache. On the other hand, ARM 11 uses a physical cache. It is to be noted that, in case of logical cache, the same physical memory area mapped into multiple virtual memory areas can be cached in different lines because the logical cache stores memory using virtual address.

ARM has two basic cache operations: flush and clean. The flush operation simply clears the valid bit in the affected cache line. The clean operation forces a write of dirty cache lines from the cache out to main memory and clears the dirty bits in the cache line. If the cache uses write-back and read-write-allocate policies the cache flush and clean operations should be done in the interrupt handler as shown in Figure 2. We can see that the shared info of the user virtual address space is cleaned and flushed before entering into the main routine while the shared buffer of the user virtual address space is sufficient with flushing. After the main routine of the interrupt handler, the shared info in the kernel virtual address space is cleaned and flushed. The flushing operation here is required to prevent the interrupt handler reading the cached shared info when it is invoked in the next time. The shared buffer in the kernel virtual address space is sufficient with cleaning.

V. PERFORMANCE EVALUATION

We have implemented RTDiP-Sync in the smc91x Ethernet controller device driver. For the experiments, we have used a pair of embedded boards equipped with Intel XScale PXA270 (520MHz) and LAN91C111 Ethernet controller. We have installed embedded Linux (kernel version 2.6.12) on the embedded boards and connected them directly without either switch or hub. In this section, we compare the performance of nonblocking TCP/IP, RTDiP-SysCall, and RTDiP-Sync. These three are the implementations of the design alternatives described in Sections III. Although the nonblocking TCP/IP does not exactly follow the bottom half based design and synchronization communication semantics, we can observe at least the performance trend of the bottom half based design with TCP/IP.

```plaintext
If RTDiP_SYNC message{
  Clean and flush user shared_info;
  Clean and flush user shared_buffer;
  Read shared_info.bufferID;
  Move the message to shared_buffer;
  Update shared_info.len;
  clean and flush kernel shared_info;
  clean kernel shared_buffer;
}
```

Figure 2. Pseudo Code of the Interrupt Handler

A. Predictability in Communication

To compare the predictability in communication of three different protocol implementations, we assume the following typical scenario for real-time distributed systems over small area networks. The sensor board periodically (for every 10ms) sends the sensing data to the application board in which traditional operating system has been installed to run several applications. The application that receives the sensing data performs a computation (for 10ms) to obtain the input values for actuators. The background applications wake up periodically (for every 10ms) and perform mathematical computation. We measure the communication overhead to get the most recent sensing data at the application (i.e., the overhead of the receive function).

The Figure 3 shows the experiment results. As we can see in Figures 3(a) and (b), TCP/IP and RTDiP-SysCall show unpredictable communication overhead. This is mainly because both use system calls that may perform signal processing and rescheduling when they return. On the other hand, RTDiP-Sync presents prominently stable communication overhead. This strongly suggests that RTDiP-Sync can provide superior predictability in communication compared with the other designs.

B. Communication Latency and Bandwidth

Figure 4(a) presents the experiment results of the communication latency test for 1B and 32B message sizes. Since the synchronization data is usually small, we present the communication latency for small messages. As we can observe, RTDiP-Sync can improve the communication latency by 23% and 62% compared with RTDiP-SysCall and TCP/IP, respectively. Although RTDiP-Sync shows better performance than the others, its latency is still around 50us. To analyze the dominant overhead of RTDiP-Sync we have measured detailed overheads on the receiver side. The results are shown in Figure 4(b). We can see that, in the small message case, the most of overhead is induced by demultiplexing and kernel buffer (i.e., sk buff) manipulation. As the user message size increases the PIO overhead to move data into the network buffer becomes dominant.
The bandwidth results are shown in Figure 4(c). As we can observe, TCP/IP achieves higher bandwidth utilization with small messages than the others. This is mainly because Nagle's algorithm implemented in TCP/IP improves the bandwidth for small size messages. For large message sizes, however, RTDiP-Sync achieves 62% higher bandwidth than TCP/IP.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a novel design of communication primitives called RTDiP-Sync for real-time distributed synchronization over small area networks. We have showed the suggested design can minimize the communication overhead and provide better overhead prediction. As future work, we intend to implement a real application or middleware over RTDiP-Sync and show practical benefits.

REFERENCES


