Abstract: Object Technology has been widely adopted and UML has emerged as a generally accepted notation for software systems development. Software Engineers are already accustomed with the OO technology and a great number of CASE tools support them in the development process. However, protocol design is still based on traditional methodologies. In the context of this paper, we present an approach that utilizes object technology and the UML notation for the development of communication protocols. A methodology is presented and a number of extensions to the UML notation are proposed to address the peculiarities of protocol design. The construction of a TCP protocol for RTLinux was selected as an example to demonstrate the methodology. Design and implementation issues are presented and the resulting system is evaluated.

1. INTRODUCTION

We have under development an Object-Oriented (OO) framework to implement the IEC61499 proposed architecture for open, interoperable and re-configurable distributed control application [1]. For the implementation of the CORFU interworking unit we selected RTLinux and we defined a modular architecture to satisfy the real-time constrains imposed by this kind of applications. The Industrial Process Control Protocol (IPCP) has been defined to satisfy real-time and non real-time requirements [2]. TCP functionality was required for the IPCP to support commissioning, configuration and on-line re-configuration of the control application. However, a TCP/IP protocol stack for RTLinux was not available. The only one found was the RTNet protocol stack [3]. RTNet provides direct access to IP-based networking from RTLinux real-time code. It implements the IP, ARP, UDP, and ICMP protocols over Ethernet. It also provides a sockets implementation for use by real-time tasks. The RTNet implementation is based on the standard Linux networking source code, with the necessary changes to make it real-time.

In [4] we have used RTNet to implement a prototype for the interconnection of a Profibus fieldbus with a Lonworks fieldbus. To satisfy the requirement for TCP functionality a first draft implementation of TCP, based on TCP Lean [5] was given, embedding it in the RTNet module. To provide a more robust, modular, expandable and layered protocol stack, we decided to redesign the TCP layer. A survey on previous works that consider the use of object-orientation and UML in the protocol design was carried out. The main directions in protocol design are briefly presented in section 2. Having applied the OO approach and the UML notation successfully in many application domains, we decided to proceed to this direction. We adapted our development methodology to address the peculiarities of protocol design. The resulting methodology, that can be applied for the design of communication protocols is presented in section 3. The design of a TCP layer for RTLinux is presented and implementation details are discussed in section 4. Finally, the proposed approach is discussed with more emphasis on performance evaluation and the paper is concluded.

2. PROTOCOL DESIGN

Current industrial practices in communication protocol design and implementation are unsatisfactory. There is a gap between state-of-the-art Software Engineering and the way communication protocols are designed and implemented. The development of communication protocols is still mainly based on the traditional procedural paradigm. The methodologies used to construct protocols can be grouped in 3 categories. According to the first category that follows the functional approach, a protocol is developed with a functional design method and a functional language that is mainly C. Although this approach results in efficient implementations, reusability and flexibility are rather poor. The second category uses a formal description technique to create the protocol’s specification, which is then translated into program code. SDL [6], Estell [7] and Lotos [8] are the most important specifications used, with SDL being the most widely adopted. Reusability in code but also in design time is also limited in this case. The third category, which is continuously gaining ground, includes methodologies that have adopted the OO approach. This approach results in implementations that exhibit increased modularity, flexibility, extensibility and reusability. Successful results have been reported by researchers applying methodologies of this category. Traditional tools have been expanded to exploit the benefits of the OO approach. Conduits+ [9] is one of them.

UML that is the standardized notation for OO, lacks many of the semantics needed for protocol engineering. Towards this direction is the work of Parsinnen et al. [10], who have developed an extension of UML, the Graphical Protocol Description Language (GPDL). They have also developed an environment aiming to translate the GPDL models into SDL models. Several researchers have also
reported their results on enhancing the OO approach, utilizing features of other methodologies. Hanish and Dillon for example have showed how a high-level Petri Net can be mapped into an OO model for use with protocol design [11]. King and Pooley have used UML to generate Petri Net models for performance prediction in a communication network [12].

3. APPLIED METHODOLOGY

For the development of the TCP protocol stack we decided to utilize our already successfully applied methodology in other application domains. However, the methodology was tailored to satisfy the peculiarities of protocol design. Furthermore, some extensions to the UML notation are proposed to better handle the communication and synchronization requirements in the protocol development domain. In this section, we briefly refer to the outline of the applied methodology giving emphasis on the modifications imposed by the nature of protocol software.

For the development of the analysis model the use case driven approach of Ivar Jacobson was adopted to delimit the system and define its functionality. Two types of models mainly constitute the analysis model of the system. The use case model and the problem domain logical view. For the use case model to be constructed, the actors that represent the roles that the software entities, which interact with the protocol play, are first identified. Each actor may perform a number of use cases. The use case, according to Jacobson, “constitutes a complete course of events initiated by an actor and specifies the interaction that takes place between the actor and the system”. In order to increase reusability from this early phase, we developed a use case diagram that captures associations between the use cases. These associations may be: adds and extends according to Jacobson or simply uses according to Rumbaugh. For the description of each use case the format proposed by Rumbaugh was adopted. However, we consider each use case in at least two levels of abstraction. The first-level description considers the interaction of the system as a whole with the entities of its environment. A more formal representation of this description is obtained using a corresponding first-level Object-Interaction Diagram (OID) [13]. In this first-level OID the protocol under development is represented as an entity and its interactions with external systems are captured. Fig. 1 illustrates the first level OID for the “active open” use case of the TCP protocol that was considered as case study in the context of this work.

This first-level OID is next expanded to a lower level OID, which we call detailed-OID. The objects that are required to compose the system for the described behavior to be provided should be identified and their collaboration defined. Coupling and cohesion are among the parameters that help the designer to identify these objects but unfortunately there are no well-defined rules to proceed. The designer’s skills are at the moment the most important parameter for an effective design to be achieved. However, the proper definition of specific design patterns from the protocol domain should increase productivity in the design of communication protocols and speed up the development process. We are currently working to this direction. Fig. 2 depicts the detailed OID of the “active open” use case. A detail reference to this OID is given in the section that deals with the TCP case study.

The problem domain model is composed of class diagrams that capture the key objects of the problem domain and constitute the logical view of the system. It is composed of objects that represent entities or concepts from the problem domain for which the system should handle information. These objects are usually used in the construction of system’s OIDs. However, a data driven approach results in the construction of the problem domain model before the use case model. A hybrid approach that involves the construction of the use case model in parallel with the construction of the problem domain model seems to be the best choice.

To proceed with the design of the system, we refine each OID to evolve to an OID that can be implemented with the selected implementation environment. During analysis, every object of the system is considered as active. However, this is not possible for the implementation. Concurrency and
synchronization must be considered and the communication and synchronization mechanisms provided by the implementation environment should be properly utilized for an optimum implementation to be accomplished.

Nevertheless, the UML standard does not provide the required constructs to capture these design issues. The mechanisms used in activity diagrams can not be used, since they refer to single thread executions, which at some point can fork and join, having only one initial and final state. In contrast, our aim is to show the synchronization between active objects that interact in the context of a use case. Moore and McLaughlin have proposed a concurrency or tasking diagram, which has been based on the collaboration diagram[14][15]. This diagram depicts the tasks and the way they interact through various mechanisms (i.e. fifos, mailboxes). These concepts are not applicable to the OIDs, so we proposed the following extensions to the UML notation to capture these semantics:

a) Concurrency. In order to be able to represent in an OID more than one thread of control, we have introduced the dashed line notation in the body of an object. When a thread of control is suspended or blocked, its body lines are converted to dashed. Automatic conversion can be obtained based on the semantics of the posted and received messages. To simplify the diagram, we decided to allow a passive object to be shown in the OID more than once. The con object for example in fig. 3 is executed by both threads i.e. the Client instance thread and the InputSegmentHandler instance thread. This can not be shown with the standard UML notation.

b) Synchronization. Operations with special semantics like set-timer(), wake-up(), wait() and notify() are defined and used to support synchronization between threads of execution.

In fig. 3, which presents a detailed level OID of the “active open” use case of TCP, the above extensions are used to introduce concurrency and synchronization into the OID.

Synchronization between the Client and InputSegmentHandler threads is obtained using an instance of Timer. The client thread which sets it, is suspended (dashed lines) until a stop signal arrives, or there is a timeout. The suspended thread is then woken-up and returns to normal execution.

4. THE TCP CASE STUDY

The Rational Rose general-purpose CASE tool [16] was used for the design of the TCP protocol for RTLinux. The use case model is presented in fig. 4. Each use case was described and a first-level OID and at least one detail-OID was constructed for it.
connection, thus making these methods non-blocking for the application.

For the implementation of our TCP layer, C++ was selected. The language is object-oriented and compilers to produce code that can be easily imported into the RTLinux kernel are available. Some programmers are wary about using C++ in the RTLinux kernel. However, even though our code utilizes many of the RTLinux’s constructs and mechanisms, like threads, mutexes, real time fifos, we have never experienced any problems due to lack of compatibility between our C++ modules and RTLinux kernel. Version 3.1 of the kernel was used for the development.

Figure 5 - Analysis class diagram of TCP (partial).

Thread communication was implemented by real time fifos (rtfifos). For the protection of shared resources we have used mutexes (type pthread_mutex_t). All Connection instances are created when the TCP module is loaded into the kernel. Dynamic allocation is not supported. The operator new and the malloc() function are allowed to be used in the RTLinux kernel only in the initialization of a module (function init_module()), since they do not satisfy real-time constraints. All mutexes and rtfifos must also be initialized (functions pthread_mutex_init() and rtf_create()) by the init_module() function.

A mutex is used to protect the table of Connections. Each Connection object also has two mutexes: one for protecting its data members associated with incoming data (i.e. input buffer, input window) and a second one for the data members that have to do with output activity (i.e. output buffer, output window). The output buffer is a FITO (First In Trial Out) buffer that we created based on [5], in order to handle partial acknowledgements (ACKs) and retransmissions when sending data.

One serious problem was that the C++ code of our TCP could not be compiled when we included several Linux header files, i.e. <linux/skbuff.h>. These headers are used by RTNet and define fundamental structs, like skbuf, which is a buffer used by Linux to store the contents of a datagram [17]. In order to overcome this, we created a separate module, written in C, to handle the interconnection with RTNet. In this way, our C++ module, named ootcp.o (Object-Oriented TCP), was isolated from RTNet and communicated only with the C module, rt_interface.o. Fig. 6 illustrates the architecture we adopted.

Figure 6 - TCP’s interface with IP

The rt_interface module communicates with RTNet directly through function calls. When rt_interface is loaded, it registers one of its functions to RTNet in order for RTNet to call this function every time a TPDU arrives. When this function is called, rt_interface passes a pointer to the TPDU to ootcp through an rtfifo. This fifo has been registered to rt_interface by ootcp, when the latter is loaded. An rtfifo handler is used to wake up the ootcp’s thread every time rt_interface writes into the fifo.

When ootcp has finished processing a received TPDU, the skbuf containing it has to be released in order to be reused. This is done by rt_interface, since ootcp can access only the skbuf’s data, that is the TPDU. Finally, upon sending a TPDU, the corresponding function of RTNet is called through rt_interface.

5. EVALUATION

Our OO methodology was tailored to the specific requirements of the protocol software and some extensions to the UML notation were proposed. The development of a protocol using the OO approach was successful. The resulting protocol implementation is easy to be understood; the detailed design diagrams mask the time consuming, low level implementation details. It is also expandable; new functionality was added first in the design models and it is then translated to code. However the most important factor was for us the performance of the resulting protocol. This is why we created a testbed in order to measure, under certain conditions, the performance of the developed protocol stack, i.e., our TCP (OOTCP) along with RTNet. We also repeated the performance measurements for Linux’s TCP/IP, using them as a yardstick. It must be clarified that our TCP has not been optimized for best performance, consequently the main
motivation of this evaluation was to verify the smooth operation of our protocol stack under heavy load.

The performance evaluation of a protocol stack involves measuring several parameters. Zanella et al [18] for the performance evaluation of TCP Westwood and Reno used simulations as well as the application of the implementations’ analytical models. They measured the TCP’s throughput with respect to variations of the error probability, buffer size, bandwidth and round trip time (RTT). Perkins and Hughes [19] have investigated the performance of TCP in mobile ad hoc networks (MANETs) by evaluating the impact of path length, node mobility and routing on the throughput of TCP. Pentikousis [20] has tested TCP Tahoe, Reno and NewReno under random and burst errors as well as combinations of these two categories. He simulated these errors during the transfer of a 5MB file between two hosts over a 10Mbps network and measured the delays inserted.

Since our protocol stack was implemented for a real-time OS, our performance tests were focused on error-less, real-time communications. The tests were conducted between two dedicated hosts, connected by crossed ethernet cable providing a speed of 100Mbps. The server application was running on Windows. On the other host was running the client application for our TCP and Linux. Once the client established a connection with the server, it received a 5MB file, stored it locally and sent it back. The results are given in Table 1.

<table>
<thead>
<tr>
<th>Bytes received</th>
<th>Linux TCP/IP</th>
<th>OOTCP/RTNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiving time (sec)</td>
<td>1.181</td>
<td>1.201</td>
</tr>
<tr>
<td>Bytes sent</td>
<td>5242900</td>
<td>5242900</td>
</tr>
<tr>
<td>Sending time (sec)</td>
<td>0.821</td>
<td>0.882</td>
</tr>
<tr>
<td>TCP throughput (Mbps)</td>
<td>51</td>
<td>47.5</td>
</tr>
<tr>
<td>Total time (sec)</td>
<td>2.002</td>
<td>2.083</td>
</tr>
</tbody>
</table>

The receiving and sending time is the time taken for each protocol stack to receive and send the file respectively. The TCP throughput [21] is calculated as:

\[
\text{BytesSent} \times 8 / \text{Sending time}
\]

It can be seen that the results are almost identical for both stacks, even though an overhead was expected from the OO implementation.

6. CONCLUSIONS
In this paper, a methodology to facilitate the development of communication protocols was presented. This methodology is a tailoring of our methodology that we have used successfully for many years in many application domains. The methodology exploits the Object-Oriented approach and the widely accepted UML notation. In our attempt to address the design issues of protocol engineering, a number of extensions to the UML notation were proposed. The methodology was utilized for the development of a TCP protocol stack for the RTLinux Real-time Operating System. The experiment was successful. The resulting implementation not only has enhanced readability, modularity, and expandability but also presents performance characteristics comparable with those of the corresponding protocol stack of Linux, even though no extra optimization techniques were used to enhance performance.

REFERENCES