Abstract—In this paper we describe two ongoing initiatives for teaching concurrency and distribution in PUC-Rio and UFRJ. One of them is a new approach for teaching distributed systems. Conventional distributed system courses follow a syllabus in which a list of topics is discussed independently and at different levels of abstractions. In Edupar’2012, we proposed a course with a novel approach, using a wireless sensor network environment to pin all topics down to concrete applications and to maintain issues such as fault tolerance and coordination continuously present. The second initiative is a smaller one, in which we insert a new topic in a Systems Software course to allow students to have a better understanding of what is application-level multitasking and of how it can be implemented. In this paper, we report on the experience of teaching the proposed syllabus and the adjustments that were necessary. We also discuss some plans for the courses in 2013.

Keywords-cross-cutting approaches, fault tolerance, event-based programming, coroutines, application-level multitasking

I. INTRODUCTION

One of the important concepts in the NSF/TCPP proposed curriculum is the idea that many topics should not be seen only at a separate, explicit, point in the curriculum but rather be treated as issues that permeate many other topics. On the one hand this may happen because many points of view are important for the understanding of an issue (for instance, architectural and programming language notions are important for understanding locality of reference). On the other hand, it may happen because one topic influences many others and cannot be studied in isolation. An important example of this is fault tolerance.

This second effect manifests itself very clearly in the conventional syllabus of distributed system courses. Topics such as fault tolerance and coordination are usually covered at a specific point of the course, often after several others have been discussed without taking these issues into consideration. With this in mind, we proposed, in Edupar’12, a more integrated approach to teaching distributed systems using wireless sensor nodes as an experimental platform on which most of the conceptual topics seen in a classical syllabus can be mapped. Because each theoretical module had an associated laboratory, all concepts were forcefully mapped to problems that were concrete for the students. Because the abstraction level of the WSN environment is very low, it does not allow the programmer to forget about the underpinning issues at any time. The course was planned for students at the advanced undergraduate or initial graduate level.

In this paper, we report on our initial results with the proposed courses, which were taught in the 2012, and discuss some adjustments that were made to the original proposal, as well as some new changes we would like to introduce in the 2013 versions.

II. DISTRIBUTED SYSTEMS WITH WIRELESS SENSOR NETWORKS

Over the last years, we have been teaching courses on distributed systems with an emphasis on programming. We have observed that the typical syllabus is a medley of important topics that are treated as if unrelated to one another, and that topics are treated at very different levels of abstraction. On the other hand, when teaching courses on wireless sensor nodes, we observed that emphasis is typically places in localization and routing algorithms, and that, although programming these devices often relies on event-based models, there is little time devoted to discussion of programming and concurrency models. From this experience, we derived the idea of teaching distributed systems using wireless sensor nodes as an experimental platform on which most of the conceptual topics can be mapped, and in which it is impossible to ignore issues such as fault tolerance and synchronization.

In the first semester of 2012 (the Brazilian “fall”), we implemented this syllabus at two universities, with different time distributions. At the Catholic University of Rio (PUC-Rio) this was a course for advanced undergraduates and graduate students, with duration of fourteen weeks. In the Federal University of Rio, this was a two-month course for undergraduate students. Overall, we found the experiences were very positive, with good feedback from the enrolled students. There were naturally some differences between the schedules that we planned originally and the ones that
were in fact implemented. In this paper, we describe the courses as they were taught and highlight some of our findings, including some issues that we would like to handle differently in the 2013 editions of these courses.

A. Original Proposal

We originally proposed that the course be structured into eight modules, which can be briefly summarized as follows:

**Module 1: Course Introduction**
- Concurrency. Threads and events. Programming with events.

Lab 1: This lab was to be the first experience of students with nesC/TinyOS [?], [?] event-based programming model.

**Module 2: Fault tolerance, Coordination and Measures**
- Coordination and synchronization. Distributed versus centralized algorithms. Synchronous and asynchronous system models.
- Physical clocks and drifting. Physical clock synchronization.
- Scalability. Experiments and analytical models.

**Module 3: Communication Patterns and Client-Servers**
- Strong and weak coupling. Communication channels (mailboxes).
- Client-server pattern. Stateful and stateless servers. Thin and fat clients. Relation to failures.
- Support for programming client-server applications. RPC and RMI. Throughput and response time.

**Module 4: Group Communication**
- Motivations for groups. Replication and peer-to-peer applications.
- Logical Clocks. Vector clocks.

**Module 5: Synchronization and Routing**
- Information dissemination. Probe/echo and heartbeat patterns. Efficiency and tolerance.

**Module 6: More coordination issues**
- Consensus. Global state.

**Module 7: More communication patterns**
- More on communication channels and mailboxes. Publish/subscribe.
- Distributed shared memory. Tuple spaces. PGAS architectures.
- Mobile code.

**Module 8: Security**
- Threats and technologies. Cryptography and authentication.
- Active and passive attacks. Denial of Service. Privacy. False Information.

B. Revised syllabus

In this section, we describe the revised syllabus, as taught in 2012. The course was taught in two versions, and we here describe the longer one, which lasted 15 weeks. During the first 10 weeks, the class met two times a week, once for a theoretical class of around one hour and a half, and once for a lab. The slides and lab handouts for the classes described below are available at http://www.inf.puc-rio.br/~noemi/inf2592/, but all materials are as yet in Portuguese.

The theoretical classes were based on materials from different sources, but we relied on a great deal of material from the book by Guerraoui and Rodrigues [?], which is naturally close to our interests because of its focus on reliability.

Most labs were exercises starting from some previously prepared code. Students were first asked to run this code and understand it, and then to modify it in steps.

1) Introduction: The first two classes were spent on the following topics:
- course overview: goals, evaluation, general structure;
- concurrency models: threads and events;
- tinynos: basic structure; components and events.

The introduction was very similar to what was originally planned. Lab #1 was devoted to implementing a simple single-node application in TinyOS, so the students would become familiar with the working environment and with TinyOS's event based structure [?]. Students were asked to study a simple application in which three leds blinked according to three different timers, and then to make a set of simple modifications.

2) Communication, Coordination, and Fault Tolerance: In the original proposal, we had planned the second module on coordination, fault tolerance, and measures, as these would be cross-cutting concerns along the course. However, when preparing the lecture, we found it would not be possible to discuss these topics without a good understanding of communication basics. So we decided to anticipate the discussion on communication and to take more time in it than planned. The following topics were discussed in this class:
Module 3. The following topics were covered:

- message passing: synchronism/asynchronism (and relation to events); identification of communication peers; communication in TinyOS.
- faults: faults and failures; dependability; common metrics;
- motivation for coordination: the need to coordinate communicating peers; common issues: message ordering, agreement, etc.;
- notation for distributed algorithms: reactive models.

The use of wireless sensor networks as our practical environment provided good material for this module. The discussion on asynchronous communication, specially as related to event-based programming, is usually a little sterile when students have no hands-on experience with events. In this case, as the class had already developed a very simples event-based sensing application, it was natural to consider the arrival of a message as an event. Faults also come naturally into the discussion, as does the notion that messages will be lost and received in different order at different nodes.

In the end of this class, we presented the notation used in Guerraoui and Rodrigues’ book [2]. Unlike many notations for distributed algorithms, theirs is quite closed to an event-based programming style, and turned out to be very easily mapped to NesC. From this point on, this was the notation used for all pseudocode in the classes.

In Lab #2, students learned to use the send/receive primitives of TinyOS. Students were asked to implement an application involving three nodes in which the nodes had to exchange messages and agree on the largest identifier among them.

3) Communication, Coordination, and Synchronization:

This module was quite similar to the originally proposed Module 3. The following topics were covered:

- coordination needed for communication;
- time and ordering;
- client/server pattern: state and stateless, thin and fat, implementation archs: pools, preallocation.

The material in this class is quite conventional. The use of interaction patterns is motivated by the need for coordinating sender and receiver activities. The client/server pattern is then discussed as the most prevalent pattern in this class. In the end of the class we discuss how send and receive operations could be implemented in an event-based system.

Lab #3 was used to show how a TinyOS application interfaces to regular computers. Students were presented to the utility programs that are available for writing applications that run on regular computers but communicate with the WSN applications, using a base station node plugged to the USB interface.. They were then asked to develop a base-station application that requested a value from a regular wireless node, which acted as a server to this client.

4) Remote Procedure Call: In the original proposal, we had not planned an entire module on RPC, but during the lab session on client/server, we thought that the abstraction would fit well with a hands-on class, and, as it is a widespread mechanism in distributed programming, we presented the following material:

- abstraction of the client-server interaction pattern;
- semantics of RPC: how much transparency do we want?
- introduction of asynchronism: oneway and futures.

This class is also an opportunity to discuss how some programming patterns are tied to execution contexts: we discussed how the RPC pattern presumes that client and server are tightly coupled. The degree of synchronism which is imposed by the RPC interaction as initially proposed makes the construction unsuitable for wide area, loosely coupled, systems. We also presented some of the critics made to the RPC paradigm by the fault-tolerance community. We then presented the asynchronous variants of RPC and how they relate to some of these issues.

In Lab #4, we gave students a complete client-server application and asked them to transform it into an RPC-style program, decomposing the client and server into stubs and application.

5) Other patterns – routing and dissemination: In this class, we turn to more symmetric patterns of interaction:

- p2p architecture;
- p2p and SPMD;
- classical interaction patterns: heartbeat and probe-echo;
- overlay and physical networks.

In this class, we first discussed different points of view about what are peer-to-peer architectures, ranging from effectively symmetric programming patterns to systems that are client-server in their architecture but are capable of using processing power at the borders of the Internet.

Although the issues of parallel and distributed programming have much in common, the two communities often use different vocabulary. We discussed how the SPMD decomposition strategy, dominant in parallel programming, can be seen as a special case of peer-to-peer architecture.

The departure from the client-server pattern again raises the issue of how to coordinate communication. We use some material from Andrew’s classical paper on patterns for interaction [3] to discuss typical patterns in this class of application.

In Lab #5, students were asked to implement a probe/echo application to collect data from the light sensors in a network. The use of the TOSSIM simulator [4] was fundamental to allow students to experiment their program on different topologies.

At this point in the course, it became apparent that the load of work from the labs was becoming excessive. Students were always doing only part of the exercises in the lab and taking the rest to complete as homework, but the homework load was accumulating. So, the next labs were used to revisit the exercises that were assigned up to this class, and
after finishing these exercises students started work on their course project.

6) Group Communication: In this module we covered some topics from Module 4 of the original proposal:

- replication and fault tolerance;
- orderings in multicasts;
- communication with different guarantees.

In this class we discussed how the basic communication issues between two nodes, seen in previous lessons, become more complex when requirements of ordering and delivery guarantees must be treated. We also introduced students to basic concepts and implementation strategies of best-effort and reliable broadcast with causal and total ordering.

7) Mutual Exclusion and Leader Election: In this module we covered some topics from Modules 2 and 5 of the original proposal:

- mutual exclusion;
- leader election;
- common algorithm patterns;
- consensus.

In this lesson, we first discussed coordination patterns for groups and some of its applications, such as mutual exclusion (with and without a centralized server), termination detection, leader election and consensus. We also revisited the broadcast algorithms and introduced students to the token circulation algorithm. Students were asked to read and discuss the sections 1, 2, 3, and 6 of the article A Fault Tolerant Algorithm for Distributed Mutual Exclusion [?].

8) Fault Detection: In this module we covered some topics from Module 2 of the original proposal:

- fault tolerance;
- types of faults.

We discussed issues related to fault tolerance in the context of synchronous and asynchronous systems, and introduced the concept of "certainty" versus "suspect" for the interpretation of failures. At the end of the class, we presented an algorithm for fault detection.

9) More communication patterns: In this module we covered some topics from Module 6 of the original proposal:

- tuple spaces;
- distributed shared memory;
- publish/subscribe.

In this class, we first presented the concept of tuple space – as implemented by the Linda language – and the main features of that communication model using message passing or shared memory. Then we discussed some issues related to the implementation of a tuple space library for TinyOS (in the context of WSN) and Java. At the end, we discussed how the Publish/Subscribe model can be combined with the tuple space concept using the TinyCOPS as an implementation example for WSNs.

The last two weeks were devoted to presentation and practical experimentation of tools being developed by doctoral students of our research group. The first one is related to a reactive programming language model for embedded systems [?], which has been properly integrated with the TinyOS programming environment. The second one offers a programming model for WSNs that integrates a set of specific components of typical WSN applications and includes facilities for dynamic reprogramming [?].

The first tool — a reactive programming language for embedded systems — offered the students the opportunity to practice a new paradigm for event-driven programming in the context of distributed computing. The second tool, on the other hand, allowed students to explore some high level programming facilities, including the reconfiguration of components previously loaded into motes.

C. Evaluation

In this first edition of the course, we held informal conversations at the end of the semester to discuss student impressions. This was specially useful because some of the students had previously taken a conventional distributed systems course. It was also possible to compare the level of interaction and understanding of the students with that of the regular course on distributed systems, as one of the authors has taught it for many years. The results were very positive: it was generally felt, both by the lecturer and by these students, that is is easier for students to think critically when presented to new algorithms and mechanisms because the communication and synchronization problems immediately come to mind, due to their programming experience with WSNs.

Besides the weekly lab assignments, the course also includes a larger final project which can be used for this evaluation. We observed that the projects fell basically into two categories. Part of the students developed sensing or acting applications for sensor nodes, and a second group of students implemented specific network algorithms. Projects in the first category typically combined and built on a great part of the concepts and mechanisms seen in class, while the ones in the second category concentrated naturally on selected lectures. This indicates that we should direct students to application-oriented projects.

We also observed that students in both categories applied the concepts they acquired in different contexts different from the ones in which they had first been presented, enforcing our impression that the integration of topics from a conventional distributed systems course with the WSN lab environment allows concepts to be more solidly understood.

We intend to develop a more specific evaluation of the understanding that students develop of the tradeoff among different degrees of fault tolerance and program complexity and performance. In the last class, when students present
their results in the final project, some of the instructors’ questions will explicitly address this tradeoff.

D. Planned Improvements

Fault Tolerance: Traditional texts on fault tolerance present very rigid solutions and hypotheses to guarantee complete reliability. In the context of WSN, as in many real distributed environments, the programmer must learn to use best effort solutions, because the cost of reliability is often too high. Although we covered some algorithms for fault tolerance, we believe we should incorporate this tradeoff along all topics seen in the course. The idea is to emphasize that middleware and libraries can offer some guarantees, but that the application must be able to deliver its service in the presence of the remaining faults. We intend to further explore this notion in the next edition of this course, classifying the fault tolerance needs of different applications and fitting solutions to these needs in the different sections of the course.

We would also like to investigate whether we can handle some common security problems as faults, allowing us to incorporate security as a cross-cutting concern, instead of leaving it as an independent module, as originally planned.

Time and Space: Most of the metrics employed on wireless sensor network applications relate to energy consumption. However, because in our case the wireless sensor network is not a goal in itself, we would like to concentrate our discussion on the number of messages exchanged and on code size and complexity. We intend to explore these metrics both in the conceptual classes and in the labs.

Lab Organization: Even with the labs starting from partially prepared code, classroom development was slower than expected, and it was often necessary to leave parts of the development to be completed at home. For our next edition, it will be necessary to restructure the lab sessions so as to balance the knowledge we need students to gain with the time schedule. We intend to explicitly organize the material for lab sessions in two parts: a first one to be done in the classroom, in which the instructor will help out with the essential new techniques and environment specifics, and a second part to be left as homework, involving less new practical aspects and more algorithmic (and intellectual) reflection.

III. CONCURRENCY: APPLICATION-LEVEL MULTITASKING

At PUC-Rio, students in the three undergraduate programs related to Computer Science (Computer Engineering, Computer Science, and Information systems) take a course on Systems Software, typically around their fourth semester. This course has much in common with conventional Computer Systems courses but it has a strong emphasis on the software perspective, and relies heavily on selected material from the book by Bryant and O’Hallaron [?]. During this course, students are exposed to basic concepts underlying computer systems, learning and exploring the ways in which these concepts provide support for implementing several abstractions provided by these systems. One of the main topics is the stack-based implementation of subroutines.

Our Systems Software course uses the IA-32 architecture as its main example and is by this time well organized into a collection of one-week modules, each of them containing a set of concepts that must be covered by the instructor and a hands-on session (see http://www.inf.puc-rio.br/˜inf1018). After introducing the basic architecture of computer systems, we cover computer arithmetic and the representation of different data types provided by the C programming language (unsigned and signed integers, floating point numbers, arrays and data structures). After that, we introduce the IA-32 assembly language, and explore the “translation” of C basic statements and control structures (assignments, conditionals, loops) to their machine-language counterparts. We next cover in depth the implementation of subroutines, including parameter passing and activation records. Stack allocation of local variables is motivated by examples using local arrays or containing function calls. Next, there is a section on interrupts and traps, in which the students learn how the operating system is activated and how interrupt handlers work. The final part of the course covers the linking process.

Our proposal was to introduce in this course some basic concepts that would allow an understanding of how concurrency can be implemented at application level. We believe this understanding is in line with the current interest in highly-parallel systems. Over the years, teaching courses on concurrent programming, we have observed that it is easy for students to have a superficial understanding of the differences between system and application-level threads, but it is hard for them to have a more concrete understanding of the issues.

A. Changes to the syllabus

We introduced an extra (class, laboratory) pair, presented to the students after they had assimilated the concepts related to subroutines. In the new topic, students learned about coroutines and their typical uses. For the theoretical presentation, we used some of the material from Michael Scott’s book on Programming Languages [?]. As an example of a coroutine library provided by a programming language, we discussed Lua coroutines [?] and showed some typical uses of coroutines implemented in Lua.

We also developed a small coroutine library in C (http://www.inf.puc-rio.br/˜inf1018/coro.tar) and devoted one hour in class for navigating with the students through its code. In this library, the same core code is shared by two different interfaces, providing symmetric and asymmetric coroutine variants [?]. The provi-
sion of these two interfaces facilitates the understanding of different uses of coroutines, such as iterators and cooperative multitasking. We then devoted a two-hour class to a lab where the students used this library to implement a very simple scheduler (http://www.inf.puc-rio.br/~inf1018/corrente/labs/lab16.html). Unfortunately, the course material is at this time available only in Portuguese.

B. Evaluation

Because we have not yet formally introduced coroutines into the course syllabus, this topic was not included in the course exams. However, because the two course instructors (Ana Lucia de Moura and Noemi Rodriguez) personally assist students during lab sessions (there are no teaching assistants), we could assess students’ understanding of the topic.

Our coroutine lab began with a very simple exercise, where students created a main program containing a loop that alternately invoked two coroutines that, at each invocation, incremented and returned the current value of a local counter. Because the students had not been previously exposed to any kind of concurrent programming, this exercise was an important asset to help students assimilate the idea of a subroutine that can suspend its execution and resume it afterwards, from its point of suspension. This understanding paved the way to the second proposed exercise, the implementation of a scheduler for an application-level cooperative multitasking application. With this exercise, students could acquire a concrete understanding of multitasking and we had an opportunity to review and discuss the differences between system and application-level multitasking, and their respective pro’s and con’s.

Several students in both classes were greatly motivated by this new topic, and excited by their results in the lab. But not all students reached the same level of understanding. However, the students presenting greater difficulty in assimilating the new concepts are the ones that presented most difficulty regarding the other topics presented in our Systems Software course.

C. Plans for 2013

We are discussing ways to promote more discussion about application-level multitasking both in the context of this course and in other courses. In the System Software course, students must develop two programming projects covering topics presented in the course. One alternative is than to have the second project that the students must hand in be related to coroutines and scheduling.

IV. Final Remarks

The results obtained in 2012 with the new approaches to teaching concurrent and distributed computing were very positive. On the one hand, we could observed weak points of the original proposals, including the time and difficulty level estimated for some of the activities. On the other hand, we could verify the feasibility of the proposals, particularly with respect to the student behavior throughout the course. Practical experience with the fundamental problems of concurrent and distributed computing motivated students to understand and apply the theoretical concepts.

Acknowledgments

This work has been partially supported by FAPERJ (grant E-26/110.320/2011) and CNPq (grant 305874/2010-1).