Dynamic resource configuration and control for an autonomous robotic vehicle

Abhishek Kulkarni, Bryce Himebaugh and Steven D Johnson
School of Informatics and Computer Science,
Indiana University, Bloomington, 47401, USA
{adkulkar,bhimebau,sjohnson}@cs.indiana.edu

ABSTRACT

This Work-In-Progress report describes an application of the 9P distributed protocol to configure and control resources on an autonomous robotic vehicle, ERTS. The vehicle was designed and developed by the participants of a graduate level course on Embedded and Real-Time systems at *Indiana University, Bloomington*. The goal of the ERTS project is twofold – to teach students about embedded system development through the interaction with the robotic vehicle and to act as a prototyping platform for researchers seeking to meet experimental objectives in areas such as computer vision, artificial intelligence, situated cognition and learning and others.

SyncFS is a synchronous, double-buffered RAM-based virtual file system that defers writes and stats to a simulated "clock edge", thus governing the asynchronous sensor and actuator components around a central common clock. The SyncFS component model allows dynamic configuration of sensor and actuator components and remote resource access for these components when running in impoverished computing domains. This model allows rapid prototyping of components on a laptop and "importing" of the autonomous cart resources on the laptop during field tests. The file servers are implemented on Linux using the 9P library implementation, npfs, and use the in-kernel 9P client, v9fs, to mount them.

We are working on a native Windows userspace file system driver to support platform heterogeneity. We plan to build on prior work on organizing sensor networks and abstracting real-time embedded systems through a file system interface and seek to extend this to integrate with higher-level navigation systems. Some of the points under active consideration include modifying 9P protocol to support aggregate communication in a sensor network, using 9P over various embedded-network protocols, and exploring alternative programming models for synchronous/reactive (SR) systems.

1. Introduction

ERTS (which stands for Embedded and Real-Time Systems) is a computer-controlled golf cart developed for and by the participants of the introductory course on *Embedded and Real-time Systems*. It was developed to demonstrate autonomous real-world navigation and serve as a research platform to students and experimenters at Indiana University. This mission imposes a need for a flexible, modular and composable architecture for rapid prototyping and faster integration of software and hardware components in the embedded system. ERTS is a reactive distributable embedded system, as the components (sensors, actuators, embedded controllers) can be placed apart and communicate with each other through a channel interface. These components maintain a persistent interaction with the environment, reacting to the inputs from the environment and responding by sending outputs to it. Using a synchronous model coupled with appropriate file system abstractions, we have developed a distibuted embedded system runtime framework for ERTS.

At the heart of the ERTS software architecture is a synchronous commit file system, SyncFS, explained in section 2. Section 3. describes the modular component framework which encapsulates SyncFS to support distributed embedded devices in the system. Finally, we discuss the work in progress pertaining to the runtime framework of the ERTS vehicle.

2. SyncFS

SyncFS is a single-writer, multiple-readers file system with synchronous, system-wide commits. The synchronizing element is a simulated clock edge modeled on the functionality of

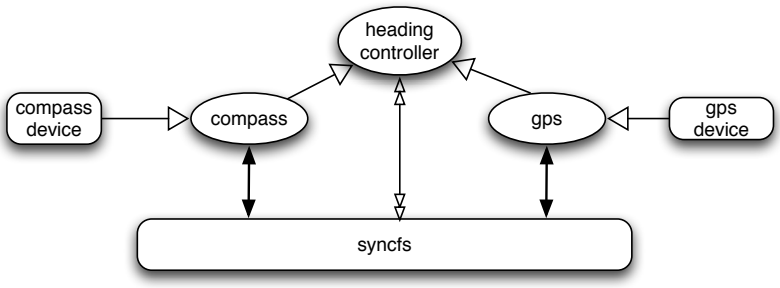

Figure 1: SyncFS component environment

globally clocked D flip-flop. In synchronous digital hardware, signals change only at clock edges. The results of these changes, in absence of propogation delay, are conceptually instantaneous. This makes synchronous systems easier to design and reason about. Several issues related to concurrency, paralellism and fault tolerance are simplified in these models. Automotive embedded systems, on the other hand, are inherently asynchronous (and so is the world around us!). SyncFS provides a synchronizing element to the component layer above it.

SyncFS supports a synchronous design model through a file-system interface. Part of this support is provided through modification to standard file-I/O (FIO) handlers. Equally important is a collection of coding conventions, governing use of files acting as communication channels. We prefer a light-weight imposition of coding conventions, if only because the design model and methods are still evolving. In other words, SyncFS is more an example of methdological support, than a end in itself.

SyncFS implements synchronization by:

1. modifying file *write* calls to defer all actual write commits until a triggering event derived from the server's physical clock.

2. modifying file *stat* calls to block the caller until all write commits are completed.

3. requiring all components to *stat* a common *clock* file at the outset of each cycle.

Under the SyncFS regimen, there are no data races, provided that all component-tasks execute within a clock cycle.A file write may make successive changes to a file without effecting its visible state to the rest of the system. On the server's virtual clock tick, the writer's version is "latched" and committed so that it becomes visible to the readers. SyncFS update their state concurrently within a clock edge. Like other reactive systems, this causes reactions to compete with each other. New inputs arrive before the end of a reaction. For explicit synchronization and to provide a common global time reference, SyncFS updates a read-only *clock* file on each tick.

SyncFS builds on the embedded file system approach of Brown and Pisupati[3], whose work was based on the original concepts prevalent in the Plan 9 Operating System from Bell Labs. The file namespace hierarchy provides a shared, language-independent region through which the ERTS system components interact. Thus, any language with standard I/O—that is, any language—can interact with system components. In class projects, for example, we use the *Python* scripting language for initial prototpying, and can incrementally convert components to C, Java, or other targets.

3. Component Framework

The component framework is a implementation-independent, conceptual model describing the components and interaction between them. Components can either be real components accesing the device through a driver, or virtual components which interface with other real components in the component ecosystem.

Figure 1 shows real sensor components like compass, GPS and virtual components like the heading controller that control the dynamics of the cart. A physical resource (sensor or actuator) in the system is represented by a textual data file that shadows the data image of the device. The interaction with these components is translated into physical transactions with the sensors or actuators if it is a real component or inter-component transactions in case of a virtual component. Components internally consist of a reactive kernel and a data handling layer coupled with the data-in and data-out interfaces.

Each component waits for a simulated clock edge, reads from its input channels, executes a function f and writes to its output channels. A component exposes a file system hierarchy consisting of the *command* file, through which the component is controlled, a *status* file, through which the state of the component can be captured. Some complex components have other auxiliary files for specialized control. Since the components model a design function behavior of a state machine, they appear as cyclic reactive processes that read/write files at the beginning/end of every cycle, performing a certain function, f. Components have input and output ports connected to other components in the environment. These ports go through SyncFS which acts as a mediator to ensure synchronous access behavior on these channels.

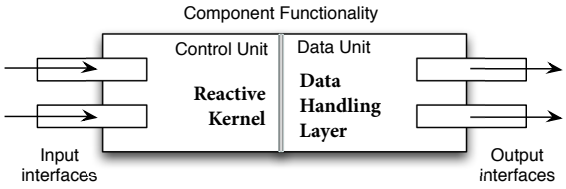

Figure 2: Component architecture

All components are peers that may access each others' resources as exposed by the SyncFS file server. In contrast to Plan 9, each component is expected to explicitly add itself to the global SyncFS namespace to interact with other components. A *terminal* component is the one that only accesses its own data.

3.1. Golf cart components

All file writes are globally synchronized when a ubiquitous system tick (a macrotick in Kopetz's terminology[2]) occurs. SyncFS provides an implicit clock component to which new and reentering components can explicitly synchronize. We have demonstrated ERTS in scenarios involving autonomous vehicle navigation. Several components for devices like compass, GPS, joystick and virtual components like bearing control, steering amplifier, configuration process and others interact to achieve autonomous navigation. The file format used by us to exchange structured data between these components is JavaScript Object Notation (JSON) for it is widely used, relatively light-weight, simple and readable.

4. Work in Progress

4.1. Windows file system support

ERTS uses the *v9fs* modules shipped with the Linux kernel since 2.6.14 to mount the SyncFS file system. Linux has primarily been the prototyping and development platform used until now. One of the primary missions of the ERTS robotic vehicles is to serve as a experimental platform for research in areas ranging from robotic vision, human-robot interaction to safety-critical systems. As we engage the cart in more collaborative research at Indiana University, we feel the need to take into consideration heterogeneous development environments and platforms. Oftentimes, the most preferred platform of development for researchers achieving experimental objectives in allied fields is Microsoft©Windows. This can be partly attributed to the lack of availability of specialized tools and software on other platforms.

This compelled us to add native support for 9P file system in Windows. We chose the approach of writing a userspace file system driver over a Windows Installable File System (IFS) driver to save us from the effort of writing a driver in the Windows kernel, and save time to focus on other important aspects of the project.

This driver is partially implemented and we hope to have full support in a few weeks. This would enable us to write components around SyncFS in Windows, and enable interaction of these components with our existing computer vision algorithms for horizon detection, visual tracking and object recognition, collaboratively developed under the hood of ERTS vision project at Indiana University.

4.2. Component Framework in Inferno

We are experimenting with implementing the components in hosted Inferno to provide us with the desired platform heterogeneity. We intend to rewrite some of the components in Limbo and use some of the abstractions provided by the language, in the form of typed channels and CSP paradigm to implement a synchronous, time-triggered embedded programming framework.

5. Future Directions

In our effort to incorporate a homogeneous resource access interface in the underlying file system, we have encountered several potential directions to explore in the ERTS project.

5.1. Programming Models for Synchronous Design Methodology

A file namespace hierarchy provides a homogeneous representation of heteregeneous resources, but it provides no explicit means of governing process synchronization and scheduling. Real-time embedded systems, safety- critical systems usually employ a synchronous design methodology through a higher level abstraction of the underlying architecture and system.

Time-triggered programming languages like ESTEREL, GIOTTO can aptly specify and model concurrent reactive systems. We intend to explore synchronous programming models and illustrate its use in real-time embedded systems using ERTS. In its current stage of implementation, we have an almost working functional interface to the SyncFS component model, written in the Scheme programming language.

5.2. Support for embedded-network protocols

As the DARPA Grand challenge demonstrated, more and more autonomous vehicles are now employing "drive-by-wire" and "steer-by-wire" technologies to off-load critical navigation functionality to a group of networked computers. We would like to push the frontier of SyncFS beyond ERTS's existing support for Ethernet/Linux network. Embedded-network protocols lay emphasis on higher data rates, low power consumption, time and event-triggered behavior. The delays due to sequential component dependencies can be reduced by implementing 9P over these protocols. Automotive systems have been using specialized bus-based protocols like CAN-bus and other fieldbus protocols for over a decade.

Realtime Ethernet protocols like Ethernet Powerline, EtherCAT provide synchronized, real-time network access over standard Ethernet. These are most likely our initial favorable targets to port 9P to minimize the inter-component communication delays.

5.3. Distributed Clock Synchronization

When used in a distributed context, there is a certain performance penalty incurred by SyncFS owing to its design. Performance and resiliency concerns thus make distributed clock synchronization desirable. This is an important goal in further development of SyncFS.

6. Conclusion

In a couple of years, commercial premium-class vehicles will contain over 1 GB of onboard system software. Autonomous vehicles are also geared with drive-by-wire capability which enforces the need for distributed, reusable component-based models for representing embedded systems. We believe the approach we have taken is quite viable for rapid prototyping in resource-rich environments. For our purposes, language independence and light-weight, composable tool chain are extremely important benefits.

7. References

1. Steven D Johnson, Bryce Himebaugh and Scott A. Dial. Homogeneous resource configuration and access for an autonomous robotic vehicle. *SAE Int. J. Commer. Veh*, **1**(1): 534–543, October 2008.

2. Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* 1997, Kluwer, Boston

3. Bhanu N. Pisupati & Geoffrey Brown. File system interfaces for embedded software development. *2005 IEEE Intl. Conf. on Computer Design* (ICCD 2005), October 2005, 232–235.