Graceful Degradation in Distributed Embedded Systems

Distributed systems provide increased flexibility, but they're also more vulnerable to failure. The goal of Carnegie Mellon's RoSES project is, "System, heal thyself."

by William Nace and Philip Koopman

As embedded systems become more distributed they provide more flexibility, the potential for greater functionality, and, unfortunately, more pieces to break. But what if there were a way to turn this proliferation of small processors into a reliability asset instead of vulnerability? What if when one component broke, the rest of the system kept working with minimal disruption? And what if we could do that elegantly, rather than just by throwing money at the problem with brute-force redundancy? We think that we can accomplish this as part of a long-term research project on the topic of graceful degradation.

A *gracefully degradable* system is one in which the user does not see errors except, perhaps, as a reduced level of system functionality — quite a contrast to most of the systems we often see. We all have heard horror stories of automobiles that need expensive trips to the shop at inconvenient times to replace $1.48 parts. Or perhaps a home security system doesn't work because a single window sensor has failed open. We like to call these systems "disgracefully degrading." These systems should still accomplish something useful — admittedly not at the same performance levels. Given a choice, we'd prefer not to call a tow truck. And the security system should still work against burglars who don't know which window to enter.

If it were simple to build gracefully degradable systems, we wouldn't have such ease finding examples of the disgraceful kind. But current practice in building reliable systems is not sufficient to efficiently build graceful degradation into any system. Rather, a few fault-tolerant computing techniques are used: 3-way (or more) brute-force replication redundancy, separate design of system responses for anticipated error modes ("failover behavior"), or attempts at multi-version redundancy. Each such approach to building reliable systems has some problems. Redundant units involve higher deployment costs, provide functionality that is only useful in the case of failure, and cannot help if the failure is systemic, such as a coding bug. Building failover functionality for anticipated failures is labor intensive — much like building the system several different times — and still leaves the user vulnerable to unanticipated failures. The last approach, multi-version redundancy, employs dissimilar redundant units in the hopes a systemic failure (design flaw, coding bug, etc.) shows up only in one of the units. This robustness technique is far too expensive for any but the most critical

Graphics Programming
ALGORITHMS
SITE LICENSING NOW AVAILABLE FOR DDJ CD-ROMs
Click Here
For More Info...
Dr. Dobb's CD-ROM LIBRARY

systems, as you're basically building the entire system with 3 (or more) different engineering teams.

The RoSES project (Robust Self-configuring Embedded Systems) is a research effort at Carnegie Mellon University focused on graceful degradation in distributed embedded systems. Graceful degradation should not be treated as a failover design problem, but instead as an exercise in designing a product family architecture (PFA). A PFA is a region of a system design space populated by different, but related, products sharing similar architectures and components. Each system instance within a PFA yields a distinct price/performance point and represents a different model in the product family. The concept of a PFA is familiar to anyone who has purchased a stereo, computer or automobile. A stereo CD component, for instance, is offered in model X with a collection of features: single disc, integrated remote, random play, etc. Model X+1 differs slightly — perhaps it has an added graphic equalizer. Model X+2 has a 5-disc carousel. All the models are very similar and were designed with the entire product line in mind. If done properly, Model X+1 is identical in almost all parts to model X. In fact it may be true that Model X has the graphic equalizer algorithms and compute resources, but just doesn't have an equalizer on the front panel. Such product line optimization has a huge impact on the profitability of the entire line, but is typically done assuming a perfectly working system rather than with an eye toward graceful degradation.

In any complex system, there may be a huge number of different system instances possible. And, if a suitable way to allocate functionality can be provided, any system in which a single component breaks can be treated simply as a closely related system in the PFA that (using a "fail-silent" assumption) just happens to differ in having the failed component missing from it. Thus, PFAs can form a conceptual framework for specifying and implementing graceful degradation within highly distributed embedded systems.

A close look at modern embedded systems shows there are some characteristics that make them particularly amenable to the PFA model, which in turn allows for automatic graceful degradation. These characteristics are: distributed functionality, smart sensors, and processing power devoted to optimizations.

Large portions of today's embedded systems are distributed collections of microcontrollers. Automobiles, especially the top-of-the-line models, typically have several networks and dozens to hundreds of network nodes. Copy machines, manufacturing machines, HVAC controls, trains, and numerous other common systems are, to a surprising degree, filled with one or more networks of microcontrollers. Because of this distributed nature, they have few single points whose failure would keep all parts of the system from operating. When a single node of a distributed network fails, the remaining microcontrollers retain enough resources to potentially accomplish many functions. The challenging problem is to allocate the remaining system resources in such a way as to achieve their full potential. As a simple example, imagine the algorithms running on a failed processor being restarted on another microprocessor, and thus keeping the system running. A more complex example might be the use of an accelerometer to calculate rough velocity should the microcontroller handling an automobile's speed sensor fail.

RoSES relies upon the increasing trend toward smart sensors (and actuators — I'll use the term sensor to apply to both) in embedded systems. In previous-generation systems, the sensors often were dedicated to a single task, and could not be shared easily. But modern sensors are built into or connected to their own microcontroller. The processing capability of the microcontroller is general — it doesn't have to operate the sensor with the originally loaded algorithm. Mobile code, or pre-positioned (but previously inactive) algorithms, allows the sensor to be put to alternate use. If the promise of microelectromechanical system (MEMS) devices becomes economically viable, sensors will have abundant processing resources. MEMS technology builds physical devices using standard IC processes, allowing integrated microcontrollers to be constructed on silicon otherwise used only for structural reasons. Smart sensors will, of course, allow for more flexibility in moving processing capabilities throughout the system and lead to more flexible systems.

Finally, RoSES exploits the tendency for systems to have an increasing number of functions that are not strictly required by the core mission of the system. In fact, much of the increasing computing power in embedded systems provides extra functionality or performance optimization rather than basic critical functions. It is often acceptable for optimization functions to be shed by the system as components fail, so long as this is done in a safe and controlled manner. For example, losing a few percent of fuel economy is often acceptable, especially when the alternative is complete vehicle failure.

Thus, there is room in many embedded systems to implement graceful degradation of functionality as a way to improve dependability for non-critical (but highly desirable) functions.

next: Reconfiguration in RoSES for Graceful Degradation

1 | 2 | 3

brought to you by Dr. Dobb's JOURNAL

© 2001 CMP Media, LLC. All Rights Reserved. Privacy Policy

HOME | DDJ.COM | ABOUT US | ADVERTISE
GO TO...

RESOURCES
Articles
Linux Handhelds
Toolkits
LANGUAGES
C/C++
Java
Ada
Erlang
Forth
Small C
Eiffel
Pascal
OS
Linux
RTOS

Reconfiguration in RoSES for Graceful Degradation

The RoSES project uses a PFA-based approach to obtain graceful degradation and other significant benefits, initially on automotive applications. The concept represents a system as a set of:

- Data flow graphs representing the connectivity among sensors, through a series of algorithms, to the actuators that affect the physical environment. I'll go into much more detail of RoSES data flow graphs below.
- System features with associated utility functions that form a lattice of acceptable systems. The lattice is a highly connected graph where each vertex represents a particular collection of features that could be implemented together and result in an associated product model. By adding or removing features, you're changing to a different product model (and thus traveling along an edge of the lattice to another vertex). See Figure 1 (pdf) for an example lattice.
- System constraints such as network schedules, or task deadlines.
- Hardware resources, including "smart" sensors, "smart" actuators, compute-server nodes, and one or more embedded networks such as a CAN (Controller Area Network) bus.

A RoSES system is a generic runtime architecture that works by providing an optimum configuration, which involves selecting a subset of possible software modules, allocating them to whatever hardware resources are available, and ensuring that the resultant system meets real time constraints without overflowing system size or bandwidth limits. In order to match standardized hardware and software components to a large variety of system configurations, RoSES uses mobile object adapters. Such adapters are flexible software interface middleware between the basic functionality of the sensor/actuator and a dynamic network object interface. In order to break the computational explosion of adapters, we allow multiple adapters to be loaded on a node to interface to logical, non-network interfaces. The role of the adapters within the RoSES system concept is illustrated in Figure 2 (pdf).

Once a particular configuration is established, a component failure (either hardware or software) triggers a system reconfiguration. The RoSES reconfiguration concept is a fairly fine-grained one, involving specific software modules/objects and potentially very small hardware components such as single sensors or actuators. The reconfiguration mechanism is, at its core, a search through different combinations of mobile object adapters for the sets that can be used on currently available hardware resources. Such combinations must be viable (supported by hardware and involving available software), meet critical system requirements, satisfy any system constraints, and provide optimal utility given available resources.

A Few Words on Reconfiguration

In today's research literature, reconfiguration is the buzzword typically used to describe the ability to choose the proper configurations to be loaded into a Field Programmable Gate Array (FPGA) operating as a co-processor. The reconfiguration we propose is a system-level reconfiguration of the software — no particular hardware techniques are required. We merely want to find the best possible use for all of the hardware of the system.

At first blush, this kind of system-level reconfiguration ought to be trivial, using current distributed system technology such as Jini or CORBA. Jini, for example, has mechanisms to discover faults (or at least allow time-outs when something fails), discover services (like sensors that provide data) available in the system, and move code across the network for execution on different nodes. While useful, these mechanisms are — in our experience — insufficient and improperly designed for the kind of embedded distributed systems that make up our target systems. Firstly, getting such resource-expensive systems implemented over an embedded network such as CAN is not trivial. Secondly, the mechanisms are useful for fulfilling local policies, but not for making globally optimal decisions.

As an example, consider how Jini uses proxy objects. A network node requiring a particular type of information would ask a Jini lookup service for an object that knows how to procure the information and, if one were available, would receive a proxy object to help the node communicate with the information source. But there is no way to guide the location service in deciding among several available proxy objects. How can you be globally optimal if you can't even tell the lookup service if you're resource constrained and would like the small object, rather than a super-whamodyne-does-everything object that is too big to fit on your node? To fulfill this need, we have instituted a centralized decision maker, called the Reconfiguration Manager.

The Reconfiguration Manager

In a system with an automatic reconfiguration mechanism, graceful degradation becomes fairly easy to accomplish. After each error is detected, a new configuration is installed to obtain maximal functionality using remaining system resources, resulting in a system that still functions, albeit with lower overall utility. Designers using such an approach do not necessarily have to examine each combination of faults to specify designated configurations, but rather rely upon a generalized reconfiguration engine to deal with any combination of faults as it actually happens. The mission of the RoSES reconfiguration manager is to fit the most useful features onto the available hardware in such a manner as to abide within all system constraints. From this statement, it is easy to see some high level requirements of the reconfiguration manager.

Available Hardware: The reconfiguration manager must somehow know what hardware is operational. This can be accomplished either by, upon notification of a fault, trimming the broken pieces from an a priori system model or to build such a model from scratch by asking each working component to describe itself.

Useful Features: The reconfiguration manager will pick the most useful adapters to put on the system. To do so, it must have an understanding of which are the most useful. To this end, some of the adapters in the system are designated as features, and tagged with a utility (i.e. usefulness) value. The utility value can be infinite for critical features that must be included in the system. Most adapters are not features, as they are useful for implementing multiple features — this concept will become clearer later in the discussion of data flow graphs.

Features are organized into classes as well. Classes are useful for expressing the fact that the same feature could be implemented in several ways. Precise spark timing control, for instance, would be implemented with a particular feature with high utility value, for instance. But an alternate means of getting rough spark control might also be possible, and thus result in a different feature with a low utility value. Now, it isn't all that useful to have both algorithms running (and in fact, could cause contention problems as they both vie for control of the hardware). So both of these features would belong to the same class, and the reconfiguration manager would know not to include both.

The manner in which features are measured is known as the utility model. The description in the paragraphs above is for the simplest utility model that will get the job done. More complex models are possible, for instance to allow synergistic interactions among features (a collection of particular features may involve more than a simple sum of utility values). The utility model's role is simple, though. It must allow the designers to communicate the desirability of different components and features to the reconfiguration manager. The reconfiguration manager needs to have a means to compare configurations to decide which is optimal. Using the utility model described above, the utility of a particular configuration is the sum of the features from unique classes.

System Constraints: Only valid configurations should be considered. A valid configuration should be schedulable (i.e. meets real-time deadlines), consistent (e.g. consumer algorithms can properly partake of producer data), and fulfill any system-specific constraints. An example of a system-specific constraint is a requirement that all the brake systems in an automobile use the same brake control algorithm.

Device Customization: Once the reconfiguration manager has chosen a configuration; it must be deployed throughout the system. Over the low bandwidth networks common to distributed embedded systems, process migration is cumbersome. Rather, the deployment will transfer small bits of state to pre-positioned executables. Or, perhaps it will bring the system down long enough to re-flash all the ROMs.

Reconfiguration Timing

The point in time when automatic reconfiguration is executed must be carefully managed. The cost of running a reconfiguration manager to determine the appropriate configuration can be significant. If the last reconfiguration was done well, there probably aren't enough slack resources (CPU or net cycles, timing slack, etc) to actually execute a reconfiguration step while the original system keeps running. Instead, we envision automatic reconfiguration employed during extreme duress or down time.

In the case of a crisis, breaking schedules to run the reconfiguration manager makes sense in that the system would be completely broken and have no chance of fulfilling its mission otherwise. Running the reconfiguration step may allow the system to find a configuration where some useful work can still be accomplished with the available resources. More typically, execution will happen when the system is down for maintenance, or at a slack time in the schedule. In an elevator, for instance, a reconfiguration step may occur during the otherwise idle time when the elevator has the doors open for passenger loading.

Eventually reconfiguration will be done on-line in real-time, but we are concentrating on providing reconfiguration as a quick-turn off-line operation to make the problems tractable in the near term. For a car example, ideally reconfiguration in response to a component failure is done while driving, but in the near-term we will instead assume that the car is pulled to the side of the road for a minute or so while reconfiguration takes place automatically (perhaps by contacting an externally available reconfiguration manager such as via an OnStar type service).

next: Operation of the Reconfiguration Manager

1 | 2 | 3

© 2001 CMP Media, LLC. All Rights Reserved. Privacy Policy

H O M E | D D J.C O M | A B O U T U S | A D V E R T I S E

GO TO...

R E S O U R C E S
Articles
Linux Handhelds
Toolkits
L A N G U A G E S
C/C++
Java
Ada
Erlang
Forth
Small C
Eiffel
Pascal
O S
Linux
RTOS

Operation of the Reconfiguration Manager

The reconfiguration manager works by solving several difficult problems simultaneously. It first has to build a data flow graph of all the available adapters. This graph is a directed, possibly cyclic graph where each vertex is an adapter and edges are the data flow between adapters. Vertices are labeled with the resources necessary for the adapter to execute — consumables such as RAM and CPU cycles as well as constraints such as microcontroller type. Edges are labeled with the bandwidth requirement and any other constraints. For instance, some communication may not be allowed on the network for lack of a defined network message type. Such edges would constrain the adapters at each end to be located in the same microcontroller. An example data flow graph for a fictitious navigation application is shown in Figure 3 (pdf). Note that meshes of similar edges have been collapsed for simplicity.

By the way, this data flow graph is a bit different from data flow compute graphs used in data-driven computations, a popular parallel processing topic from the 60s and 70s and now part of most industrial-strength compilers. Such computations build a graph of all the operations the data would have to flow through to get a correct answer for the computation, much as our data flow graphs. In embedded systems, however, the data keeps getting generated by the sensors and flowing through the graph. This is a result of two phenomenon of embedded systems — cyclic state and time-triggered architectures. Many embedded systems, often as a result of rotating machinery, proceed in a cyclic manner, repeating computation periodically. For example, spark plug timing calculations are accomplished every time the camshaft in an engine rotates. Often, repetition is a result of a time-triggered architecture, where each task is accomplished periodically rather than, as in an event-triggered architecture, as a result of particular events occurring. In a time-triggered architecture, for instance, the loss of a particular message is of less consequence, as it will be regenerated during the next computation period.

This data flow graph has a fair amount of redundancy in it. Different sub-graphs produce the same outputs, and thus only one (at most) need be implemented. Each configuration is a different combination of graph vertices. We can eliminate configurations with vertices that don't contribute to a sensor-to-actuator path. We would also like to eliminate configurations that don't fit within the constraints of the hardware and network. This determination is actually an NP-hard (i.e. "really, really tough") problem known to computer theorists as "bin-packing." The reconfiguration manager tries to find a way to pack the vertices into bins (the microcontrollers) with fixed size (the resources of the microcontroller).

Several of the vertices in the graph are designated as features, as discussed earlier. Features are additionally labeled with class and utility values (other utility models would require different values, or means of

SITE LICENSING NOW AVAILABLE FOR DDJ CD-ROMs
Click Here For More Info...
Dr. Dobb's CD-ROM LIBRARY

computation). The utility of a particular configuration is the sum of the features from unique classes in the configuration. Recall that some classes of features are critical and must be included in any final configuration.

The preliminary reconfiguration manager we are implementing works by searching through all feature classes for the maximum utility set of features. It then prunes the data flow graph of those adapters (vertices) that cannot help provide data for the features chosen. Finally, it searches through alternatives, checking each set to see if it will pack onto the available hardware. In the case of failure, it tries different collections of adapters or goes back and selects different sets of features. Once a configuration is chosen, mobile code or animation of pre-stored code is used to load the chosen adapters onto the proper hardware. We expect to use bin-packing heuristics to do this in a matter of a few seconds for a large system.

Reconfiguration Management Is More Than Just Graceful Degradation

Graceful repair reintegration (and graceful upgrades): Ultimately, it is important to gracefully reintegrate a repaired component as well as to reconfigure in the face of a component failure. As subsystems are repaired or replaced, the reconfiguration manager determines configurations that can use the added resources to restore functionality. In addition, reconfiguration allows access to configurations beyond the original product design. If a repair is made with a replacement part that has superior performance, reintegration of the repair part is not just a repair, but also a system upgrade. Beyond that, it is possible that new components (and associated abstract functionality blocks and software modules) can be added to perform field upgrades using the same approach as that employed for reintegrating repair components.

In fact, a key insight is that graceful degradation and upgrade via reconfiguration are simply ways of moving down or up the lattice of points in product family architecture. When some hardware breaks or is inserted, it's as if a different model in the PFA had been realized. The reconfiguration manager then can determine the best collection of features to install on available hardware.

Reconfiguration as Logistical Support: The reconfiguration mechanism can also be exploited to provide a potentially major logistical benefit — parts replacement with non-exact spares. Doing so frees maintenance personnel from the burden of carrying every conceivable spare part. For example, they might just carry more capable, and expensive, generalized spares instead of cost-optimized specific repair parts. (But reduced labor costs for trips back to the shop to pick up just the right spares could easily offset any increased component costs.) In emergencies, sub-optimal repair parts might be used to perform temporary partial repairs. While the military implications for compact spares inventories and non-exact battlefield repairs are obvious, such issues are also important for any system involving mobile maintenance personnel or systems with few installations served per supply depot.

In addition, a major cost of supporting legacy systems is the need to provide legacy spares. In the U.S., vehicle OEMs are required to keep a spare parts pipeline available until well after the average lifetime of the vehicle, subjecting the OEM to an interesting factory utilization challenge. The OEM must weigh the warehousing costs of spare parts with the need to keep a factory line hot for the parts. This mandate will be increasingly challenging as more and more automobile subsystems involve digital electronics — entire chip fabrication processes may need to be kept operational far beyond their obsolescence merely to provide spare parts designed a decade earlier.

An automatic reconfiguration mechanism may ease such logistic nightmares. Rather than replace a part with its exact duplicate, a non-exact spare may be employed. The reconfiguration mechanism can then be used to find a different configuration that still provides for the same level (or perhaps an enhanced level) of functionality. Legacy spares may be provided by building updated sensors and actuators to hold several different algorithms just so they can be used in older systems.

Reconfiguration Problems

Reconfiguration is not a panacea, otherwise it would already be in widespread use. Challenging areas, beyond just the technical issues behind building a system this way, that need consideration include: validation/certification, debugging/technical support, multi-vendor coordination, and the assignment of blame (or legal liability) when something doesn't work.

The certification issue revolves around ensuring a certification authority (e.g., FAA or NRC) is comfortable with the reconfiguration mechanism and the manner in which configurations are chosen and deployed. Hopefully, once the agency is comfortable with a particular mechanism, then recertification of future product models becomes a simple process. If reconfiguration is deemed too great a risk, one can employ a separation strategy where the safety-critical functionality is partitioned away from all other features and apply reconfiguration only to the less safety-critical parts. Such separation is common, for instance, in vehicles where one network is employed for critical engine control or braking, and another network is used for the less critical power windows, door locks, and so on.

The flexibility of a reconfigurable system may cause concern among system designers and developers who desire strict control over the system. Determining proper system operation is difficult for even a single configuration of a complex system. A proliferation of configurations raises another level of complexity for the debugging process to overcome. This debugging problem (like so many other complexity problems) may be alleviated with adherence to a carefully controlled architecture. In the same way that the abstraction of an object-oriented system reduces overall complexity and assists with interface compatibility, reconfiguration is much easier when the adapters fit well-defined and properly abstracted logical interfaces. In addition, the reconfiguration manager must scrupulously log all configuration changes and make the configuration data available to the problem resolution team.

In a multi-vendor environment, reconfiguration runs into additional issues. Recall that reconfiguration is, at core, a process that allocates available resources to functionality. Well, what happens when the resources come from one vendor's unit and are used to provide functionality for unit from a different vendor? The first vendor may object as the cost to provide the resources makes the unit more costly compared to any non-reconfigurable competitors. In addition, liability for an accident in such a situation is unclear. A jury could easily find either of the unit vendors, the integrating vendor or the vendor providing the reconfiguration algorithms at fault.

Conclusion

Automatic reconfiguration mechanisms have a lot of potential to achieve graceful degradation in distributed embedded systems. A reconfiguration mechanism optimizes the system by choosing functionality from a product family architecture. The chosen functionality is then carefully positioned on network nodes where computing resources are available. We hope this idea will elegantly provide for both graceful degradation and significant logistical benefits, although it does present some challenging research issues as well. We will report on our future progress at http://www.ices.cmu.edu/roses. We are grateful for support of this research from Bosch Electronics and the General Motors Satellite Research Laboratory at Carnegie Mellon University.

Bill Nace is a Ph.D. candidate who enjoys watching what happens when systems break. He has collected many such experiences in his 14 years of service in the US Air Force.

Phil Koopman uses his many "war stories" from industry experiences to help him teach an advanced embedded system design course at Carnegie Mellon University.

1 | 2 | 3

PERFORCE SOFTWARE No red tape getting Perforce Software Configuration Management. Click here.

© 2001 CMP Media, LLC. All Rights Reserved. Privacy Policy