An Approach to Preserving Sufficient Correctness in Open Resource Coalitions

Orna Raz
Institute for Software Research, International
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213 USA
+1-412-268-1120
http://www.cs.cmu.edu/~ornar/
orna.raz@cs.cmu.edu

Mary Shaw
Institute for Software Research, International
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213 USA
+1-412-268-2589
http://www.cs.cmu.edu/~shaw/
mary.shaw@cs.cmu.edu

Abstract

Most software that most people use most of the time needs only moderate assurance of fitness for its intended purpose. Unlike high-assurance software, where the severe consequences of failure justify substantial investment in validation, everyday software is used in settings in which occasional degraded service or even failure is tolerable. Unlike high-assurance software, which has been the subject of extensive scrutiny, everyday software has received only meager attention concerning how good it must be, how to decide whether a system is sufficiently correct, or how to detect and remedy abnormalities. The need for such techniques is particularly strong for software that takes the form of open resource coalitions – loosely-coupled aggregations of independent distributed resources. In this paper we discuss the problem of determining fitness for purpose, introduce a model for detecting abnormal behavior, and describe some of the ways to deal with abnormalities when they are detected.

Keywords

Medium-assurance software, everyday software, fitness for task, fault tolerance, open resource coalitions, sufficient correctness, software homeostasis, distributed component-based software

1 Introduction

Most software in everyday use is not "correct", yet it supports a wide variety of useful work. We consider some of the special problems that arise when software relies heavily on external resources outside the control of the developer. We present an approach to working with software that is "sufficiently correct" – fit for its intended use. We focus on ways to maintain sufficient correctness even when the external resources change or malfunction.

Changing technology has enabled a new computing paradigm based on coalitions of network-based resources. We focus on this setting because, even more than most software, these coalitions are subject to forces outside the user's control. The usual problems of software system fragility are exacerbated in this setting: system reliability is vulnerable to network and resource reliability and the likelihood of problems increases with the number of independent resources, especially since they are managed independently and used without commitment for support.

Previously we described *open resource coalitions* [19] and introduced a framework for studying fitness to task (*sufficient correctness*) and resilience to operating abnormalities (*software homeostasis*) [20]. Here we explore the problem of achieving software homeostasis, with emphasis on techniques for detecting abnormalities, or excursions from normal operation. We introduce a model for system degradation and the correspondence between user-visible failures and underlying faults, and we propose an approach to detecting abnormalities and remedying them.

2 Everyday software

We are interested in software that does not require high-assurance guarantees, for example because interactive use allows human oversight or the cost of failure is low. Such everyday software needs only everyday assurances. This widens our options for implementation and analysis techniques. For example, everyday software might

- Include fallible information such as reviews, experience, historical record and reputation in analyses.
- Deal with abnormalities by detection and repair rather than prevention.
- Use imperfect parts.

Prior work has studied ways to handle inconsistencies during system development (with the objective of eliminating the inconsistencies before system release [7, 8, 17]), or in execution (treating them as exceptional conditions [2] or using requirement/design information [10]), or in specific domains[6].. Here we are interested in constructing useful software systems from components that are likely to be fallible in operation and are outside our control. Our approach is generally in the class that Nuseibeh calls "amelioration" [17].

2.1 Open Resource Coalitions

The recent exponential expansion of the Internet, especially through the access mechanism of the World-Wide Web, enables a new set of architectural opportunities. The Internet hosts a wide variety of *resources*: primary information, communication mechanisms, real-time data feeds, invokable applications, control that coordinates the use of resources, and services such as secondary (processed) information, simulation, editorial selection, or evaluation.

These resources are independently developed and independently supported. Client software created by or for a specific user can invoke these resources as components in a software system. In stark contrast to traditional closed-shop development, the development mode here is aggressively *open*-shop, with constituent parts remaining under control of their developers. Often, the client's use of a resource is not known to the resource; indeed, the client may rely on incidental properties of resources that the proprietor of a resource has made no commitment to supporting.

The resulting systems are vulnerable to unannounced changes in the underlying services, so they are more appropriately regarded as *coalitions* than as systems. The selection and composition of resources is likely to be done afresh for each task (or even dynamically during processing), as resources appear, change and disappear. These software assemblages are *open resource coalitions* [19]. They are open in a sense even broader than that of Das and Fekete [5], who study systems in which components are designed to operate in many different environments: their setting requires active cooperation among the components to achieve a distributed commit protocol.

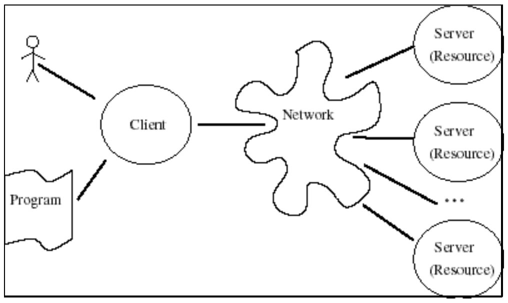


Figure 1. Architecture of open resource coalitions

Figure 1 shows the typical architecture of an open resource coalition. The user is served by a local client that invokes resources over a network as well as local code to produce results. A variety of invocation protocols may be involved. For simplicity, we consider resources available over the Internet via World Wide Web protocols. Issues of everyday utility and fitness to task are particularly pertinent for these coalitions, because the architecture is intrinsically vulnerable to faults outside the developer's control. These problems are exacerbated when information from resources is further processed, rather than simply viewed by a human.

2.2 Sufficient Correctness

As noted above, everyday software often behaves imperfectly, yet we manage to get useful work done. Two factors affect our tolerance for abnormal behavior. First, we are usually less concerned about abnormalities when we are confident that we'll notice the problems and try again or fix the problem. Second, we are usually less concerned about abnormalities when their consequence is small. As Figure 2 suggests, these two factors establish a space in which we can differentiate the significance of various abnormalities.

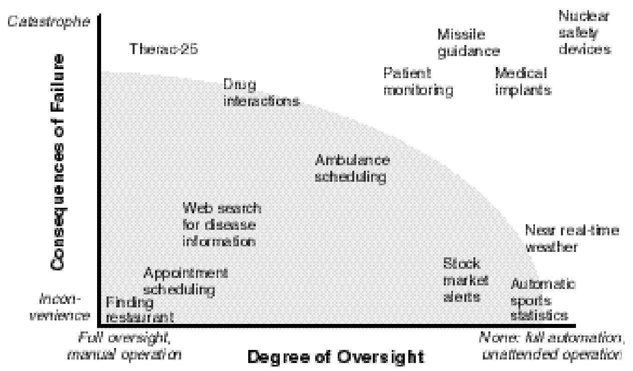


Figure 2. Region of everyday software

Everyday software lies in the shaded region of Figure 2 For everyday assurance we ask the question, "Is it good enough for the use I intend?" Moreover, we are interested in gaining assurance about fallible systems built from fallible parts. Improving the quality of the individual resources is an interesting question, but a different one.

The intrinsic uncertainties of open resource coalitions, especially network performance and independent management of resources, make such coalitions inappropriate as an architecture for the critical systems in the upper right corner of the space. However, the shaded area of Figure 2 contains many examples for which responsible risk management may find the cost of gaining full confidence higher than the cost of detecting and repairing failure. For such systems, we are interested in *sufficient correctness*: whether the system can be trusted to do what we intend to use it for, and do it well enough for practical purposes. The result of the analysis should be an envelope of allowable behavior that is captured in the coalition's credential [18] so that it sets the standard for initial and ongoing validation.

In addition to simply asking whether useful results can be obtained from imperfect systems, we investigate ways to make component-based systems less fragile than their individual constituent components.

3 Software Homeostasis

Homeostasis is the propensity of a system to automatically restore its normal, or desired, or equilibrium state when something occurs to upset or disturb that state. ***Software homeostasis*** *as a* *software system property refers to the capacity for monitoring system behavior and dynamically modifying the system to repair abnormalities, or deviations from expected behavior [20].*

The user is interested in the results that the system delivers, so our model addresses end-to-end behavior. For everyday software, a user often doesn't need exact results, but rather expects results that lie in some envelope of normal operation. A user can also cope with some degree of degraded operation, but if even that lower expectation is not realized, the system is broken. We begin with a qualitative model to develop intuition and discuss the problems associated with making the model and state definitions more precise.

3.1 Normal and abnormal operation

The traditional model of a software system recognizes only correct and incorrect behavior. The traditional model often assumes that a specification of correct behavior actually exists. Such a traditional system is not influenced by external events, so if it ever breaks, it stays broken. Figure 3 illustrates the major states of such a system: It begins operating in normal state; during a given time interval it has some probability P_{NB} of breaking, and if it ever breaks it remains broken. The longer the program runs, the more likely it is to become (and remain) broken

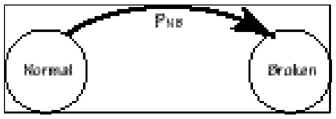


Figure 3. Traditional model of software failure

More complex systems often have more complex criteria for acceptable behavior, They are often specified only incompletely or informally. In addition to the set of normal behaviors, they may remain somewhat useful in a degraded state. The degradation might, for example, involve performance, precision, or even parts of the desired information. In addition, real-world systems are often influenced by external events. For example, distributed systems degrade or break when network service is disrupted, and they often return to normal service when the network problems are repaired. Similarly, the independent resources that contribute to a coalition may experience service interruptions that are repaired by the resources' proprietors. Following Arora and Kulkarni's model [1], we can consider introducing an explicit state corresponding to degraded operation.

We are interested in these more realistic systems, which are "biddable" in Jackson's sense [14]. Figure 4 shows relations among the major states of these systems. They still break during a given time interval with probability P_{NB}, but they also degrade with probability P_{ND} (and subsequently break, of course, with probability P_{DB}). In addition, system problems may be repaired through external events (spontaneously, from the standpoint of the system) that induce transitions P_{BN}, P_{BD}, or P_{DN} as shown in Figure 4.

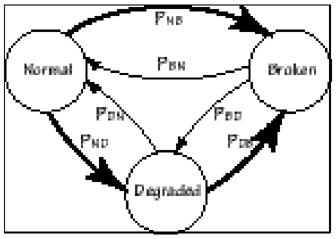


Figure 4. Degradation and failure in ideal systems

Unfortunately, Figure 4 assumes not only precise specifications of normal and broken behavior, but also explicit distinctions between normal and degraded behavior. For everyday software it may be more useful to take the view suggested by Figure 5. This view makes only soft distinctions among the states, emphasizing instead that transitions may degrade or improve performance and that there is a sometimes-fuzzy distinction between working and broken. It suggests adding transitions in all operating regions to deter or repair deterioration. In doing so it sacrifices the simplicity of the state model for analysis, but it also shows that much of the apparent simplicity is artificial.

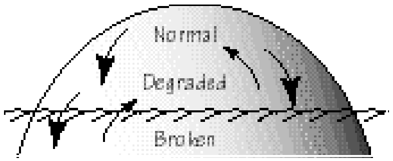


Figure 5. Degradation and failure in real systems

3.2 Achieving homeostasis

Adopting the view of everyday software in Figure 5 sets the stage for adding healing mechanisms to systems. In the shaded region of Figure 2, the cost of preventing failure may be much greater than the cost of coping with degraded service or even failure. The need for healing mechanisms is particularly great for open resource coalitions, where the usual fragility of software is compounded by the use of remote resources without commitment for support.

To achieve homeostasis in practice we must: (a) establish (even informally) the regions of normal, degraded, and broken operation, (b) identify ways to detect or deter the faults; (c) add mechanisms to the implementation that will tend to restore or preserve normal operation.

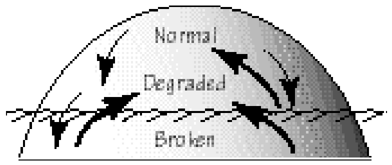


Figure 6. Effects of homeostatic mechanisms

Adding mechanisms that monitor system behavior and take remedial action as necessary has the effect of altering the transition probabilities to better favor preserving or restoring normal behavior, as suggested by the stronger upward transitions in Figure 6. These mechanisms include both deliberate repair and automatic housekeeping.

3.2.1 Defining acceptable regions of operation. Preserving sufficient correctness in a system depends on recognizing normal, degraded, and broken states. Ideally, these states would be formally specified. Practically, however, this is unlikely: specifications are difficult and expensive to develop, and they are intrinsically incomplete [18]. They will rarely be justified for everyday systems.

A more realistic option is to use information that we can reasonably expect to be (or become) available. Fortunately, the amount of information available increases through the life of the system. It can come from many sources:

- Whatever specifications do exist, including
  - information provided by self-describing resources (e.g. XML)
  - credentials provided by a third party covering functionality, quality, etc.

  These specifications will usually be incomplete, and information may come in different formats and different precisions for the various resources in a system.
- Semantic redundancy among resources (duplicate resources, overlapping resources, or resources with different treatment of comparable content) or within resources (internal consistency, consistency of change over time)
- Historical information on individual resources, such as:
  - data collected on the behavior of a resource as used in a particular coalition (e.g., by the coalition)
  - data collected on the wider behavior of a resource (e.g., by its proprietor or a third party)
- Historical information on the coalition itself, including interactions with the user about its fitness

Even if the original specifications are quite informal, usage statistics can be collected and interpreted over time, thereby refining the definitions of the states. We are investigating statistical techniques, especially with user feedback, for this approach. Techniques such as Ernst's [9] for dynamically detecting the de facto invariants are also promising.

We do not rely critically on a precise, explicit distinction between normal and degraded operation. User understanding of everyday systems is largely informal, so we must handle distinctions that are informal and often qualitative [4]. Even so, partial information can be enough to improve overall behavior. Further, many of the mechanisms of interest serve both to preserve and to restore health.

3.2.2 Precision of transition estimates. The usual level of specification in an everyday system does not support high precision in failure estimates. This is especially true for resource coalitions that depend on incompletely specified external resources. However, even an inexperienced developer can make qualitative, even rank-ordered, estimates. It follows that analysis techniques must be able to work with ordinal-scale data until enough information is available to support ratio-scale analysis [3, 4, 11]. We can nevertheless consider the likelihood of transitions that improve or degrade system behavior.

The probabilities of transitions in system behavior should be composed from the probabilities of individual faults. As more data becomes available, the individual estimates, as well as the overall estimate, can be refined. An initial refinement would convert the estimates to be quantitative. Further refinement would improve their accuracy. This could be done through observations of the system's behavior (self-monitoring along with user-provided refinements, due to better understanding of the system), or with information and specifications provided by a third party.

With refined estimates, it seems plausible to predict overall system behavior with a Markov model derived from the state diagram of Figure 6. The initial model might simply be an educated guess, and progressive improvement might result from applying machine learning and inference techniques (e.g. Bayesian methods or max likelihood).

3.2.3 Detecting abnormalities. Homeostasis can be realized through a combination of ongoing maintenance and explicit detection and repair of abnormalities. The former is especially important when the operating regions are defined informally. In any case, some abnormal cases will be known explicitly. For these it is useful to detect failures or impending failures. Such detection mechanisms are a major focus of this paper, and we deal with them in more detail in Section 5.

Following standard usage [15], a software failure is a result that violates the specification or an unexpected software behavior observed by the user. A software fault is the identified or hypothesized cause of the software failure.

Our model of system health represents failures as seen by the user. The actual mechanisms that restore or preserve normal operation handle faults in the implementation. To do this, we must (a) identify the types of faults that cause transitions between system states; (b) establish mappings between failures and faults, and (c) use multiple techniques to detect faults. Section 5 elaborates on these points.

3.2.4 Restoring and preserving normal operation. When abnormal (degraded or broken) behavior is detected, the homeostatic mechanisms should operate to restore normalcy if possible. Recovery mechanisms include:

- Find another route to the same data (e.g. mirroring)
- Use a format converter (e.g. for format problems)
- Retry (e.g. for connectivity problems)
- Use an alternate resource (e.g. with equivalent data)
- Compute the result another way (e.g. use several other resources from which the result can be inferred)

If it is not possible to restore normal operation, the user may continue work with degraded service, for example:

- Extrapolate from prior data (e.g. guess current temperature based on a model of normal change together with recent data)
- Accept degradation of service (e.g. reduce amount of data, accept lower performance, update less frequently)
- Proceed without missing information (e.g. when trying to find best price, proceed based on available bids)

In Figure 7 we classify faults according to their origin in the system's architecture. Such a classification is useful for understanding what recovery mechanisms are applicable. For example, a fault originated at the client cannot be treated by alternating source (a server related recovery).

Recovery is achieved via a combination of automatic, semiautomatic, and manual intervention. Automatic recovery requires no user intervention. Semiautomatic intervention consults the user for approval before taking suggested actions. Examples include (a) the user may wish to preserve the option to reject certain actions the system is capable of automatically executing (such as installing plug-ins); (b) the system may list several options for the user to choose from. Manual intervention requires the user to think and possibly to supply additional information or to take action. Examples include (a) the system may issue a warning indicating a possible problem, and the user may decide whether this requires action; (b) the system may require additional data or authorization from the user (e.g. to register with a resource and authorize payment).

Many recovery techniques (plus housekeeping mechanisms that run routinely instead of in response to faults) can also be used during normal operation to prevent excursions from that state. For example, if the system detects that the content of a web page has recently changed radically and that the new page contains a redirect, no failure will appear to the user. Nevertheless, it is often prudent to update the coalition's URL to the target of the redirect instead of waiting until the redirect page expires and the link breaks (thereby losing the redirect address).

4 Our case study: noncritical health support

The IWSSD case study sketches a teleservices and remote medical care system (TRMCS) [13]. This study emphasizes high-assurance services, including reliability of a distributed system, privacy, etc. There are important differences between the underlying design objectives of the case study and the underlying objectives of our work in preserving sufficient correctness in open resource coalitions:

- The case study emphasizes safety-critical assurances, but our interest is in everyday assurances for everyday problems.
- The case study assumes that a system will be developed with closed-shop methods, then installed for the user as a turnkey package. Our interest, however, is in enabling developers to create systems from existing resources that serve their own particular needs.

We share other assumptions of the case study, and our variant retains these characteristics: the distributed character of the system, the need for data fusion, and the existence of some requirements for performance and availability

Since we are interested in a significantly different point in the design space, we discuss a variant of the case study. In place of the critical-response scenario, we consider support for an individual with a combination of stable chronic conditions and a personal interest in health and fitness. We sketch a distributed system that will provide this individual with information, advice, reminders, and loosely-coupled monitoring by a medical support office. Details of this adapted case study appear in the Appendix.

5 Detecting abnormalities

We present an approach for detecting degraded or broken operation. The approach involves classifying faults, mapping between faults and failures, detecting faults with a variety of techniques and making tradeoffs. To illustrate this approach, the Appendix walks through detection scenarios in our variant of the case study.

5.1 Fault classification

The user's view of the system, and hence the operating regions, is expressed in terms of visible failures (or their absence). Those failures will be consequences of specific faults in the system implementation. Figure 7 shows places in the resource coalition architecture where faults can occur. We classify faults primarily according to their origin in the system: client, network server, or resource.

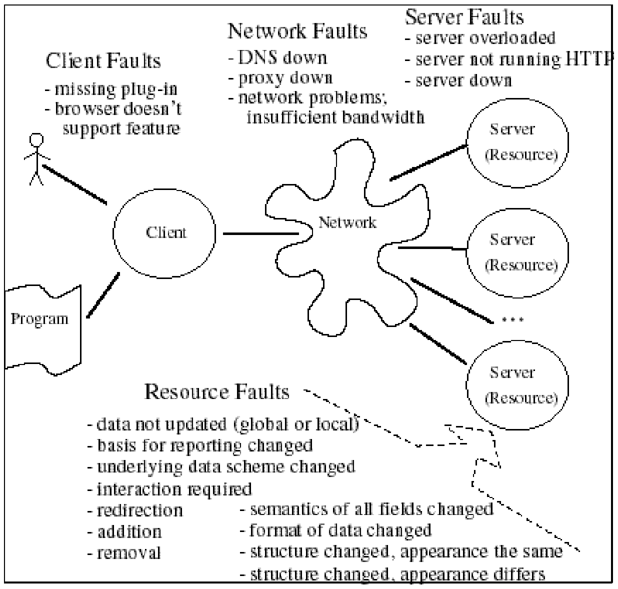

Client Faults
- missing plug-in
- browser doesn't support feature
Network Faults
- DNS down
- proxy down
- network problems; insufficient bandwidth
Server Faults
- server overloaded
- server not running HTTP
- server down
Client
Program
Network
Server (Resource)
Server (Resource)
...
Server (Resource)
Resource Faults
- data not updated (global or local)
- basis for reporting changed
- underlying data scheme changed
- interaction required
- redirection
- addition
- removal
- semantics of all fields changed
- format of data changed
- structure changed, appearance the same
- structure changed, appearance differs

Figure 7. Sources of faults that create failure

The system failure space can be divided into levels, producing the following hierarchy:

- Connectivity: "Can I get any data at all"?

- Syntax (structure and format): "Can I parse the data?"
- Semantic: "Does the data make sense?"

Network faults cause connectivity level failures (Figure 7). Client faults cause syntax or semantic failures. Server faults cause mainly connectivity failures, whereas resource faults cause syntax or semantic failures.

Faults do not always cause failures. Frequently, however, they degrade the overall health of the system (Figure 5). Since this gradual slide can eventually lead to failure, it is often appropriate to repair even faults that did not lead to visible degradation or failure.

5.2 Fault-failure mapping

To implement mechanisms for failure prevention and recovery, we must establish a correspondence between the user-level failures and the faults that arise in the implementation. Tables 1 and 2 give examples of such mappings.

Failure / Fault	Performance: slow connection	ERR failed DNS lookup	ERR503 service unavailable	ERR host unknown, unable to locate host	ERR file contains no data	ERR connection. refused
DNS down		X	X	X		
Proxy down			X	X		
Server down			X	X		
Server not running HTTP			X	X	X	X
Network problems, Insufficient bandwidth	X		X	X		
Server overloaded	X		X	X		

Table 1. Connectivity failure – fault mapping

The mapping between faults and failures is not unique. Indeed, a failure may be caused by a combination of faults. Fortunately, we only need to identify the underlying fault precisely enough to initiate recovery. The same recovery measures may apply to multiple faults, so fine discrimination between faults is often not necessary. For example, if unreasonable output values are detected, it is possible to map the problem to one of several groups of possible causes (data consistency, interaction, redirection, addition, removal, semantic, format, structure). The class of data consistency failures (changes in basis for reporting or in the underlying data scheme) could be handled by accepting degradation of service (accuracy in this case) or by switching to a semantically redundant resource.

Some failures can be detected easily. These include performance problems and failures announced by explicit error messages (e.g., HTTP protocol-level errors). Other failures, especially semantic failures and some format and structure related failures, may be very hard to detect. To do so may require continuous monitoring of a resource's behavior, possibly along with monitoring semantically redundant resources. Semantic failures, in particular, are often defined in application-dependent terms.

Statistical techniques such as machine learning techniques can reveal trends and discrepancies. Initial experiments indicate that information retrieval techniques (word frequencies and document distance metrics) are useful for detecting unexpected content. User feedback is required for improving the definition of the normal behavior envelope, especially for semantics. This would improve accuracy, and lessen the amount of user intervention for failures.

Failure / Fault	Unreasonable output values	Unreasonable output types	Combination (values and types)	Some output value that should be updated stays fixed	all output values that should be updated stay fixed	program breaks (hang, abort, crash)	ERR400 bad request	ERR401 unauthorized	ERR connection refused by host	ERR404 page not found	ERR bad file request	ERR missing plug in	Parts of page not displayed
Data not updated at all					X								
Some dynamic data not updated				X									
Basis for reporting changed, e.g. counting rules)	X		X	X	X								
Underlying data scheme changed.	X		X	X	X								
Interaction required (e.g. need to login)	X	X	X			X	X	X	X				
Redirection	X	X	X			X							
Addition (e.g. provide more services)	X	X	X			X							
Removal (e.g. provide less services)	X	X	X			X							
Semantic of all fields changed (e.g. different service)	X	X	X			X	X			X			
Format of data changed (same content)	X	X	X			X							
Html structure change, appearance the same (e.g. CSS)	X	X	X			X							
Html structure change, appearance change	X	X	X			X	X			X			
Missing plug in												X	X
Browser doesn't support feature											X		X

Table 2. Semantic and syntax failure – fault mapping

5.3 Fault detection

The client's objective is end-to-end delivered service. Faults can defeat this objective in many ways, and there are correspondingly many ways to detect and repair faults. Point solutions to many problems of connectivity and syntax exist now, but the semantic level is largely unexplored.

Some examples of faults, means for detecting them, and remedies are:

- *A server is overloaded.* This is detected by performance problems at the client that are unique to that server. Switching to a mirror site may solve this problem

- *The client is missing a plug-in.* This is detected by a browser error message or a dialog offering to download a plug-in. The problem is solved by installing the plug-in, possibly after requesting user approval, or by deciding to accept degraded service without the plug-in
- *Local connection to the network is lost.* This is detected by inability to get data from any server. In some facilities, a facility staff handles network problems; in these cases the client should usually keep retrying until the server is up again. In other facilities, network connectivity is the responsibility of individual users; in these cases manual intervention may be needed.

Detection techniques are diverse, not only due to the diversity of faults, but also due to the diversity in the sources of information available for determining the normal and degraded states. For example, some detection techniques, especially at the structure/format and semantic levels, may be based on monitoring. Monitoring is done based on the available information, yielding the following flavors:

- Monitoring for adherence to explicit specifications (e.g., server availability [21])
- Monitoring semantically redundant resources to detect when one behaves significantly differently from the others (e.g. it gives a nutrition value that is much lower than the other resources give)
- Tracking a single resource against its prior history (e.g. notice what parts of the data historically change, and make sure new data is changed accordingly; monitor response time of a server and notice degradation)

The applicability of detection techniques and their accuracy depend on the fault type and the information available. The research challenge is to find a framework for integrating individual point solutions and to add solutions where there currently are none, especially at the semantic level.

5.4 Design, synergy and tradeoffs

Detection techniques differ in type (the kinds of faults they can detect), effectiveness (the likelihood they will detect a fault), requirements (e.g., cooperation of servers, availability of similar resource) and cost/performance.

Using the techniques in isolation will yield scattered and incomplete results. More comprehensive detection requires coordination among techniques. In addition to coordination for coverage this must take into account cost-benefit tradeoffs. For example, one technique for detecting a semantic fault might be effective but costly (e.g., large data and bandwidth) whereas another might be less effective, but also less costly. The choice would depend on the user's perception of the incremental benefit of the former.

Using a combination of techniques based on several information sources may enable detection of problems that each of the techniques alone is unable to detect. These can be new classes of problems, or finer granularity in detection within some class of problems. Combining techniques may have additional benefits, if information is accumulated over time. It seems promising to apply statistical techniques to accumulated data to find patterns that typically precede faults. This could be especially helpful in detecting semantic and syntax abnormalities as they occur.

Combining and integrating results from different techniques and information sources requires careful attention. Different techniques may conflict with each other, and the entire suite of possible techniques is likely to be overkill in any specific application. Users should be able to select the parts that are relevant to their problems. They should also be able to determine the cost or effectiveness of detection.

It follows that for appropriate cost-effective detection, we need to take into account cost-benefit tradeoffs, performance overhead, effectiveness of combining techniques, advantage of information accumulated from several detectors, and from several sources of information, and conflicts arising from combining techniques

6 Discussion

We have described work in progress, including a model and preliminary design for adding homeostasis to interactive software systems, particularly open resource coalitions. We now sketch our plans for developing these ideas.

6.1 Quality of Semantic Service

We see a potential for making statistical guaranties to the client. To do that, we would need to model the quality of the overall service delivered to it. Models exist at the connectivity level for Quality of Service (QoS), usually related to network resources availability. We envision a model for *Quality of Semantic Service (QoSS)*, to enable guarantees at the upper levels. Both detection and recovery involve tradeoffs, so dimensions along which statistical guarantees can be made remain to be defined.

6.2 Fault tolerance and Markov models

To show that a system's overall health is better with homeostatic support than without, it seems feasible to adapt fault-tolerance models and to use Markov models. Huang et al [12] argue that the most important dimensions of fault tolerance, from the user’s point of view, are availability and data consistency of the application. The emphasis on each dimension varies in accordance with the nature of the application. They observe that most applications have modest degrees of requirements along these dimensions, but the trend is to (cost effectively) increase those degrees. They put the application software layer on top of the hardware and operating/database system layers. Then they use an end-to-end argument to claim fault tolerance is needed at the application level.

To adapt this model from closed applications to our open model, we place the semantic level on top of the connectivity and syntax levels. The important dimensions become connectivity and semantic consistency. We can place a system without homeostasis along these dimensions and check to see whether the added mechanisms contribute towards higher semantic consistency with an acceptable increase in performance overhead. If we can create a Markov model for our system (e.g., based on [16]), we can also verify that the limiting probability of being in the broken state decreases.

6.3 Validation plans

To evaluate our model and an implementation we must show that the added homeostatic mechanisms improve the system's overall health. This entails

- Showing effectiveness of detection mechanisms, including adaptive techniques for refining specifications.
- Showing effectiveness of the individual prevention and repair mechanisms.
- Showing that the framework makes suitable provisions for coordinating multiple mechanisms, including cost-effectiveness tradeoffs.
- Estimating costs of detection and repair in benchmark scenarios.
- Estimating benefits of preventing system failures and comparing those to the costs of detection and repair.

6.4 Applicability to high assurance software

It's tempting to think that the homeostasis model applies to high-assurance software as well as everyday software. After all, adding monitoring to prevent excursion from normal operation is a reasonable safety feature. We suspect that the model does apply in a general way, but that doing so will require different treatment of details. For example, the states of interest are almost certainly different: the high assurance case surely requires more precise specifications of acceptable states. Our model assumes that the developer has no control over individual components – high assurance applications require guarantees about these parts and about the protocols that invoke different parts. Finally, most high-assurance techniques assume closed-shop development, and hence control over all the components.

Acknowledgements

This research is supported by the School of Computer Science at Carnegie Mellon University. We thank Phil Koopman and Shawn Butler for their advice and support.

References

1. Anish Arora and Sandeep Kulkarni. Component based design of multitolerant systems. *IEEE Tr. Software Engineering*, vol 24, no 1, Jan 1998, pp. 63-78.
2. Robert Balzer. Tolerating Inconsistency. *Proc. ICSE-13: 13th Int'l Conf on Software Engineering*, May 1991, pp. 158-165.
3. L. Briand, K. El-Emam, and S. Morasca. On the Application of Measurement Theory in Software Engineering. *Empirical Software Engineering*. vol 1, no 1, 1996.
4. Shawn Butler, Somesh Jha, and Mary Shaw. When Good Models Meet Bad Data: Applying Quantitative Economic Models to Qualitative Engineering Judgments. *2nd Workshop on Economics-Driven Software Engineering Research (EDSER-2)*, May 2000.
5. R. Das and A. Fekete. Modular Reasoning about Open Systems: A Case Study of Distributed Commit. *Proc. 7th Int'l Workshop on Software Specification and Design (IWSSD-7)*, Dec 1993, pp. 30-39.
6. C. Dellarocas and Mark Klein. An Experimental Evaluation of Domain-Independent Fault Handling Services in Open Multi-Agent Systems. *Proc ICMAS-2000, The Int'l Conference on Multi-Agent Systems*, 2000.
7. Steve Easterbrook. Learning from Inconsistency. *Proc 8th Int'l Workshop on Software Specification and Design (IWSSD-8)*, Mar 1996, pp. 136-140.
8. Steve Easterbrook et al. V&V Through Inconsistency Tracking and Analysis. *Proc 9th Intl Workshop on Software Specification and Design (IWSSD-9)*, Apr 1998, pp.43-49.
9. Michael D. Ernst et al. "Dynamically Discovering Likely Program Invariants to Support Program Evolution". *Proc. ICSE '99: 21st Int'l Conf on Software Engineering*, 1999, pp. 213-224.
10. M.S. Feather et al. Reconciling System Requirements and Runtime behavior. *Proc 9th Intl Workshop on Software Specification and Design (IWSSD-9)*, Apr 1998, pp.50-59.
11. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous & Practical Approach*, International Thomson Computer Press, 1997.
12. Yennun Huang and Chandra Kintala. Software Fault Tolerance in the Application Layer. In Michael R. Lyu (ed), *Software Fault Tolerance*, Wiley 1995, Ch 10.
13. Paola Inverardi and Henry Muccini. IWSSD10 Case Study, Teleservices and Medical Care System. *In this volume*.
14. Michael Jackson. Software Requirements & Specifications, Addison-Wesley, 1995.
15. Michael R. Lyu. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, 1996.
16. M.T. Mainini. Reliability Evaluation. Ch 10 of M. Kersekn, F. Sagleitti, eds, *Software Fault Tolerance: Achievement and Assessment Strategies*. ESPRIT, Springer-Verlag 1992.
17. Bashar Nuseibeh. To Be and Not to Be: On Managing Inconsistency in Software Development. *Proc 8th Int'l Workshop on Software Specification and Design (IWSSD-8)*, Mar 1996, pp.164-169.
18. Mary Shaw. Truth vs Knowledge: The Difference Between What a Component Does and What We **Know** It Does. *Proc. 8th Int'l Workshop on Software Specification and Design*, Mar 1996
19. Mary Shaw. "Architectural Requirements for Computing with Coalitions of Resources". Position paper for *1st Working IFIP Conf on Software Architecture*, 1999.
20. Mary Shaw. "Sufficient Correctness and Homeostasis in Open Resource Coalitions: How Much Can You Trust Your Software System?" *4th Int'l Software Architecture Workshop (ISAW-4)*, May 2000.
21. D. Slama, J. Garbis, and P. Russell. *Enterprise CORBA*. Prentice Hall, 1999, Ch 16.

Appendix: A resource coalition for noncritical health care

For reasons described in Section 4, our intended domain differs from the premises of the case study in assumptions about degree of assurance to be provided and in locus of control of software development. We therefore sketch different requirements before presenting a solution.

A.1 Domain: helping individuals marshal useful resources

As discussed in Section 4, individual users have access to a wealth of on-line resources, but they lack means of combining these resources in useful ways and of determining whether a coalition they construct is fit for its intended purpose and whether it will remain fit. Issues of initial construction and assurance are discussed elsewhere [18,20] and are the subject of other research. Here we consider the "will remain so" part of the problem.

The following scenario illustrates an everyday system that requires everyday assurances:

> *Pat has a combination of stable chronic medical conditions (e.g., diabetes) and a personal interest in health and fitness. Pat's interest in fitness is motivated in part by a desire to manage the medical conditions through diet and exercise rather than medication. Pat has an "exercise buddy", Lou; they schedule joint exercise outings. Pat reports medical status and a diet and exercise summary daily to a service that provides long-term monitoring, advice, and alerts if Pat's condition becomes unstable. Pat also orders medical supplies on-line; the prescriptions must periodically be renewed by the medical service. Pat wants to be notified -- very selectively -- of pertinent news, and he would like access to a support network. These resources must be integrated so that information needed by one resource is provided automatically when it's available. Pat needs to be able to access this integrated support at home, at work, and while traveling. To achieve this Pat builds an open resource coalition. This is not a high assurance system --- since Pat's condition is stable the coalition needs to supply medium assurances. A temporary failure is tolerable, provided the system resumes normal (or possibly degraded) operation in a timely manner, as required for Pat's medical monitoring.*

We assume that the user has the means to access all the networked resources, use them under program control, acquire the locally-running software, and compose the elements into a tasteful and functional whole. This example explores what additional mechanisms can be introduced in the coalition to monitor system health.

An online version of this example, with live links, is available at http://levers.compose.cs.cmu.edu/orca/ex/iwssd.htm

A.2 Available resources

To serve Pat's needs -- and Pat's shared needs with Lou -- we select from resources available on the World Wide Web (WWW) in May 2000 and add speculative resources to round out the design. Table A1 gives a sampling of these resources.

Kind of resource	*Existing resources*	*Speculative resources*
Information resources	Nutritional and fitness reference [24]; Reference material for medical conditions; Weather [25]and other information [26] affecting outdoor exercise; Personalized news feed [27], personally filtered news feed [28];	Barcode reader to read food packaging, together with information resource that retrieves nutritional information given barcode information
Scheduling resources	Personal calendar [29, 30]; Shared calendar [31, 32];	
Computation resources	Exercise and diet log Calculators [33]; Nutrition Analysis Tool [34]; Diet Analysis Tool [35];	
Monitoring resources		Proprietary medical monitoring and notification service
Local code	Diabetic Daily Log [36]	Feed diet log information to medical monitoring Schedule exercise based on calendar and weather

Table A1. Available resources

A.3 System Requirements

A.3.1 Functional requirements

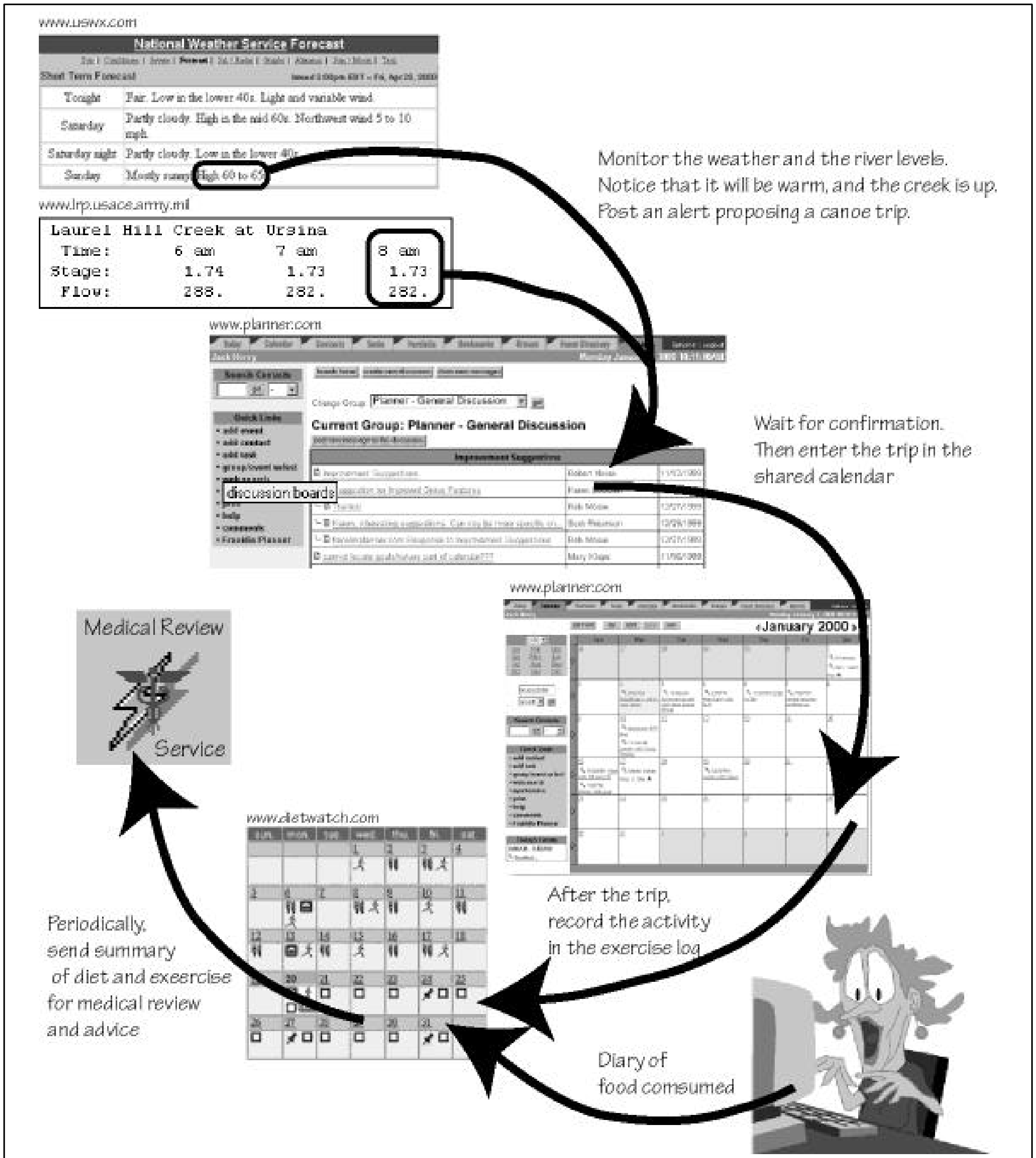
- *Data logging:* Allow Pat to record diet, exercise, and medical status information, either manually or by various sensors
- *Medical monitoring*: Medical center review of data logs, returning advice and automatically reordering medical supplies
- *Exercise buddy coordination:* Shared calendar, coupled to forecast weather conditions

A.3.2 Performance and reliability requirements

- *Timing:* No special requirements; everyday web performance will suffice
- *Availability:* Minor service interruptions tolerable, but

connectivity must be good enough for status monitoring. Delay even as long as several hours in communications with the medical center are tolerable, but delivery must be guaranteed.

- *Data persistence:* Exercise, diet, and medication logs must be reliable in recording and preserving information and in delivering it when requested.



www.uswx.com
National Weather Service Forecast
Short Term Forecast
www.lrp.usace.army.mil
Monitor the weather and the river levels.
Notice that it will be warm, and the creek is up.
Post an alert proposing a canoe trip.
www.planner.com
Current Group: Planner - General Discussion
Improvement Suggestions
discussion boards
Wait for confirmation.
Then enter the trip in the shared calendar
www.planner.com
‹January 2000›
Medical Review
Service
www.dietwatch.com
After the trip, record the activity in the exercise log
Periodically, send summary of diet and exercise for medical review and advice
Diary of food consumed

Tonight	Fair. Low in the lower 40s. Light and variable wind.
Saturday	Partly cloudy. High in the mid 60s. Northwest wind 5 to 10 mph.
Saturday night	Partly cloudy. Low in the lower 40s.
Sunday	Mostly sunny. High 60 to 65.

```
Laurel Hill Creek at Ursina
 Time:        6 am       7 am       8 am
Stage:         1.74       1.73       1.73
 Flow:         288.       282.       282.
```

Figure A1. Use case scenario for noncritical health care

A.3.3 Privacy and security requirements

- *Medical review center:* Communication must be reasonably secure. The level of sensitivity is that of personal medical data and transmission of medical prescriptions.
- *Other resources*: Resources should be pre-selected to have acceptable privacy policies. At present, the available resources don't even provide secure connections -- but the information exchanged with these resources is only mildly sensitive.

A.3.4 Level of automation requirements

The purpose of this coalition is to organize a set of related information flows. As much recording as possible should be automated, though it's not possible to automate all of it. Most of the work of moving information from one place or format to another should be automated. Actual decisions must, of course, remain in the hands of the user.

A.4 Use case

Figure A1 illustrates one use case of the scenario described above. Here we imagine how the coalition might work; in the next section, we discuss ways the coalition might achieve homeostasis.

Pat and Lou want to go canoeing whenever weather and river level permit. Pat builds a coalition to coordinate the services of the indicated resources. Except for the medical review service, some form of the resources currently (May 2000) exists at the locations indicated. The resources are depicted with snippets from their WWW interfaces. The user interface of the actual coalition must, of course, be much more consistent and better integrated.

In this use case, the coalition monitors weather [25] and river [26] conditions. Here it notices that conditions are right for a canoe trip. It proposes a schedule to Pat and Lou, notifying them via their shared calendar/planner [31]. Pat and Lou both confirm this activity, so the coalition schedules the trip, again in the shared calendar/planner. After the trip, Pat provides details of the trip, and the coalition records it in Pat's diet/exercise log [33]. Pat also enters nutritional information in the diet/exercise log every day or two. Twice a week, the coalition sends summary information for medical review. This use case does not include replies from the medical review. Pat is personally involved in only three steps: (1) accepting the proposed trip, (2) providing details on the trip, and (3) recording what he ate. The first is an essential human oversight function. The latter two reflect current lack of technology to collect the data. It's easy to see, though, that a wireless barcode scanner for food labels and a resource that converted the bar code to nutritional information could be added to the coalition.

A.5 Healing scenarios

Looking at the system's architecture, as depicted in Figure 7, we identify many potential faults that can cause failures at the connectivity, syntax or semantic levels. For each of the failure levels, we present one scenario of failures in the operation of the use case coalition, detection of these failures and recovery from the underlying faults. In addition, we present scenarios of background maintenance that is independent of any detection process.

A.5.1 Connectivity level

- *Fault:* The server running the weather resource might be overloaded. This might result in a standard HTTP error message, or in performance degradation – a very slow connection.
- *Detection:* The coalition can detect HTTP error message by using an existing third party service [23]. It can detect performance problems by monitoring the rate at which data is received and noticing significant slowdown. The coalition can examine several of its resources to distinguish between network problems and server problems. If connection problems exist with all of the coalition's resources, then this is identified as a network problem. Otherwise it is identified as a server problem. Notice that the fault here can be transient or permanent. Furthermore, degradation in the connection might be an indicator that the server is about to go down.
- *Recovery:* Accumulated data may be useful here. If the weather server currently being used frequently experiences load problems, it might be wise to use a mirror, or switch to a different weather resource. This is also the appropriate recovery if the problem is permanent. Although this strategy is reasonable for stateless transactions like weather, switching calendar services requires additional mechanism to keep current data in both the main and the backup calendar formats. If the connection is slow and likely to be transient, there are several options: accept the degradation, use cached data, temporarily switch to a mirror site, or use a different resource. In the meanwhile, retry the original resource periodically and switch back when it is up again.

A.5.2 Syntax level

- *Fault:* The river level resource might change the format of its data from text to postscript. This might result in local code being unable to parse this data, issuing an error message. (It might also output unreasonable values, turning the failure into a semantic level failure.)
- *Detection:* The coalition is unable to extract data ("scrape the screen"), so the river level monitoring portion of the site breaks.
- *Recovery:* Use an existing file format conversion tool [22].

A.5.3 Semantic level

- *Fault:* The nutrition calculator resource (used for foods consumed) might change its underlying data scheme. This might result in incorrect nutrition value for particular foods.
- *Detection:* This data inconsistency problem can be detected in several ways, each relying on different available information. (1) Using data gathered on the behavior of the resource performing previous calculations, the coalition might compare the current result to previous results and notice a divergence. (2) Monitoring semantically redundant resources, all calculating the same nutrition values, the coalition can compare their outputs. A problem is indicated when the output of the main calculator is significantly different from the others. (3) If semantic specifications are available, monitoring the data may reveal that the results are not within the specified range.
- *Recovery:* This may be only a change in accuracy or perhaps a correction of a previous problem – the user must decide whether this is a real problem. If this is indeed a problem, but only accuracy is hindered, degradation in accuracy of calculation may be acceptable. It may be possible to extrapolate from prior data. Switching to a different resource is another option.

A.5.4 Detection-free recovery

Homeostasis may be achieved not only through detecting specific problems and recovering, but also by performing maintenance and repair activities regularly. This can take place at all levels; we present one for connectivity and one for semantics.

The system may observe that Pat often consults the weather and river levels in the morning, whether or not the coalition suggested a trip. To protect against unavailability of this information because of connectivity failure, the coalition retrieves and caches this information several times a day. If live information is not available when Pat requests it, the coalition can deliver slightly stale information. This slightly degraded performance is usually acceptable.

Pat's accumulated exercise and nutrition information would be very time-consuming to re-create, and Pat cannot afford full backups of his home system. The coalition identifies the information that can't be derived from other data and does regular backups of that selected information. The coalition might even make the backups at a remote site.

Appendix References

22. Typed Object Model (TOM) Server, http://wheel.compose.cs.cmu.edu:8001/cgi-bin/browse/objweb
23. Spyonit Site Monitor Service, http://www.spyonit.com/Add?_spyid=sitemonitor
24. The Merck Manual of Diagnosis and Therapy, http://www.merck.com/pubs/mmanual/sections.htm
25. Weather information, http://www.uswx.com/us/wx/PA/021/
26. River level, http://wmw.lrp.usace.army.mil/current/yc.html
27. Infobeat, personalized news feed, http://www.infobeat.com/
28. Spyonit, http://www.spyonit.com/Home
29. Yahoo, personal calendar, http://calendar.yahoo.com/
30. Excite, personal calendar, http://reg.excite.com/mps/login?pname=planner&pstr=Excite+Planner&brand=xcit&targeturl=http%3A%2F%2Fplanner.excite.com%2F
31. Franklin planner online,, http://www.planner.com/
32. Super calendar, http://www.supercalendar.com/
33. DietWatch,, http://www.dietwatch.com/
34. Nat tools for good health, http://www.nat.uiuc.edu/
35. Diet analysis web page, http://dawp.anet.com/
36. Diabetic daily log, http://members.aol.com/kennzo/tddl.htm