Research Issues in No-Futz Computing

David A. Holland, William Josephson,
Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein
Harvard University
Ada Lim
University of New South Wales

Abstract

At the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII), the attendees reached consensus that the most important issue facing the OS research community was "No-Futz" computing; eliminating the ongoing "futzing" that characterizes most systems today. To date, little research has been accomplished in this area. Our goal in writing this paper is to focus the research community on the challenges we face if we are to design systems that are truly futz-free, or even low-futz.

1 Introduction

The high cost of system administration is well known. In addition to the official costs (such as salaries for system administrators), countless additional dollars are wasted as individual users tinker with the systems on their desktops. The goal of "no-futz" computing is to slash these costs and reduce the day-to-day frustration that futzing causes users and administrators.

We define "futz" to mean "tinkering or fiddling experimentally with something." That is, futzing refers specifically to making changes to the state of the system, while observing the resulting behavior in order to determine how these relate and what combination of state values is needed to achieve the desired behavior. When we refer to "no-futz" computing, we mean that futzing should be allowed, but should never be required. We interpret "low-futz" in this way as well.

It should be noted that reducing futz is not the same as making a system easy to use. It is also not the same as hiding or reducing complexity: it is about *managing* complexity and *managing* difficulty. Computer systems involve intrinsically complex and difficult things. These are not going to go away. The goal is to make it as easy as possible to cope with that complexity and difficulty.

Systems can be easy to use but still require unnecessary futzing: TCP/IP configuration on older Macintoshes was easy to adjust, but was difficult to set properly. One can also imagine a (purely hypothetical) system that hides all its complexity: it appears to need almost no futzing at all, until it breaks. Then, extensive futzing is required, to figure out what happened.

The goal of No-Futz computing is to eliminate the futzing due to poor design or poor presentation, not to try to find a silver bullet for software complexity; no-futz computing attacks areas that are needlessly complicated, not those that are inherently complicated.

Let's begin with an example of a good, hi-tech, low-futz device, and understand its basic characteristics. While reading the rest of this section, keep in mind the computer systems you use regularly (particularly the ones you dislike) and how they differ from the example.

Our Xerox 256ST copier is a no-futz device. It performs just about every function imaginable for a copier: it collates, staples, copies between different sizes of paper, will copy single-sided originals to double-sided copies and vice versa, etc., and it even sits on the network and accepts print jobs like a printer. However, it demands no futzing. It has instructions printed on the case that describe how to accomplish common tasks. Its user interface makes it impossible to ask it to do something it cannot. It keeps track of its operating state, continuously monitoring itself, and communicates in a simple fashion with its operators. When there is a problem it can diagnose, it displays a clear message on its console. (For example, "Paper tray 2 is empty.") When it detects a problem it cannot diagnose, it begins a question-and-answer dialog with the operator to diagnose the problem and determine an appropriate course of action - and then, in most cases, it guides the operator through that course of action. The questions it asks are simple, and can be answered by a novice, such as "Did paper come out when you tried to copy?" The key factors that make this device no-futz are:

- Ease of use: The user documentation and user interface are organized in terms of the user's tasks, not in terms of the system's internal characteristics.
- It is unusual to encounter a situation where it is not clear what to do next, even in the presence of various failures.
- Self-diagnostics: When a failure occurs, the copier diagnoses it and offers instructions for fixing things.
- Simple, clear communication: It never asks the user a question that the user cannot answer.

What makes this such an interesting example is that only a decade ago, photocopiers required much futzing, mostly by expert servicemen, and were extremely frustrating for all concerned. Since then, not only have copiers become vastly faster and more powerful, but both the use and maintenance of them has become vastly easier. Today's copiers have one-tenth the components of their predecessors, significantly more functionality, and dramatically reduced futzing [6]. How can we make similar strides forward in computing?

That which works for a photocopier may not be suffi-

cient for computers: the copier is a relatively straightforward device with well-defined function and state, whereas general-purpose computer systems have a wide variety of functions, have essentially infinitely mutable state, and are subjected to complicated and often ill-understood interconnections both within themselves and with other computers.

In the rest of this paper, we first discuss some current approaches to futz reduction, arguing that these do not attack the problem directly and have negative side-effects. We then discuss how futz arises in computer systems and describe what we believe is the key to a real solution: understanding and managing system state. Then we outline some directions for future research, discuss briefly some existing related work, and conclude.

2 Current Approaches to Futz Reduction

The cost and frustration associated with futzing has led to three common approaches to futz reduction: (1) limiting the scope of functionality, (2) homogeneity, and (3) centralization. These approaches are not mutually exclusive and are frequently used together.

The copier described above is an example of the first approach: it is a special-purpose device. Relative to a general-purpose computer, its functionality is quite limited. In this context, it has addressed the futz problem quite well. Since futzing involves state changes, special purpose systems, which have relatively limited state spaces, can offer correspondingly reduced futz. Other low-futz, limited scope devices include dedicated file servers (e.g., Network Appliance's filers) and special purpose web or mail servers (e.g., Sun's Cobalt servers) among others.

Homogeneity is the second approach to futz avoidance. This approach is most often seen in large installations. In order to reduce total installation-wide futzing, a single standard machine configuration is deployed everywhere. If there is a problem, any machine can be replaced with any other machine. Systems can be reinstalled quickly from a master copy. Maintenance requirements are reduced drastically. Custom management tools need only interact with one kind of system, and are thus much cheaper to build. The administrators see the same problems over and over again and can prepare solutions in advance; nobody besides the administrators needs to futz with anything.

This approach can reduce global futz drastically; however, it does not address the underlying problem: the amount of futzing required by a single machine is constant. Furthermore, it has other flaws: first, it is inherently incompatible with letting users control their computers. While this is fine or even desirable in some environments (e.g., the terminals bank tellers use), it is unacceptable in others (e.g., research labs). Second, it is a security risk. The same homogeneity that makes system administration easier also makes break-ins and virus propagation easier: if you can get into one system, you can get into all of them the same way [1]. Third, most organizations grow incrementally. Adding new computers to a collection of identical existing ones is difficult: the new ones are rarely truly identical, which inevitably cuts into the economy of scale.

The third approach to futz reduction is centralization. Centralization moves state and its accompanying requirements for futzing, away from the systems with which people interact directly and into places where it is more conveniently managed. This gives administrators tight and efficient control over each system. This makes it more convenient for system administrators to futz and lets system administrators do more of the futzing and users less of it. While this does reduce cost, there is no actual reduction in total futz. For that, another approach is required.

These three approaches are capable of reducing the futz of, or at least the cost of maintenance for, computer systems and networks. However, all of them are limiting and/or have negative consequences. This is a result of attempting to reduce the total futzable state, instead of the futz problem directly. We advocate the direct attack.

3 The Source of Futz

One definition of "futz" is in terms of state manipulation. Thus, the more state there is to manipulate, the more futzing a system allows. Mandatory futzing arises when it is not clear by inspection or documentation what manipulations are required or when the supposedly correct manipulations fail to produce the correct result. At this point, one must experiment (or call for help).

If one can manipulate the system state without resorting to experimentation, futzing has not occurred. For instance, seasoned Unix administrators do not have to futz to add accounts to their systems. But beginners generally do. And even seasoned administrators usually have to futz to get printing to work.

Note that the degree of futz depends on the level of expertise of the user. A premise of no-futz computing, however, is that one should not have to be an expert, or the cost of being an expert should be quite low. Unix systems are already quite low-futz for hard-core experts, but it takes years and years of apprenticeship to reach that level. Reducing futz for a select few is not a solution, so we need to examine sources of futz as they appear to a casual user.

The mutable state of a computer system can be broken down into the following categories (this may not be a complete list):

- **Derived state:** State automatically derived or generated from other state.
- **Policy state**: Configuration state that reflects policy of a site or user.
- **Autoconfig data**: Data to be served in some manner by the system in order to enable autoconfiguration for other systems. For example, /etc/bootptab.
- **Cached state**: Cached results from autoconfiguration protocols.

- **Manual config state**: Configuration state that reflects the setup of the operating environment or hardware, and needs to be set manually.
- **OS file state**: files (programs or data) that are part of the operating system, as well as their organizational meta-data.
- **Application file state**: files (programs or data) that are part of installed applications, as well as their organizational meta-data.
- **User file state**: user files and their organizational meta-data. For example, a secretary's word processor files, or web pages.
- **Application context**: persistent saved application state that is not user data. For instance, many environments try to automatically recreate on startup where you were when you left the last time.
- **System context**: persistent OS state that is not in any of the above categories. For example, file system meta-data.
- **Cryptographic keys**.

Policy state is a source of futz: the system acts on its policy settings, and if it acts incorrectly, somebody needs to tinker with the settings until it behaves properly. Unfortunately, policy state cannot be avoided in a general-purpose computer system: policy decisions need to be made by humans and the computer needs to know what they were. One can reduce futz in this area by cutting back the amount of state, and building special-purpose systems, but that inherently reduces the amount of functionality as well. Reducing futz in this area without cutting back functionality is feasible as we outline in the next section.

Autoconfig data is another source of futz. This category reflects futz that has been "centralized away" from other systems. It is not necessarily the case that all autoconfig mechanisms require a server to serve data, but many of the existing ones do. It is not unreasonable to suppose that development of more sophisticated autoconfiguration can reduce or eliminate most of the state and thus the futz in this category.

Cached state is not normally a source of futz. Cached results can be purged or updated as necessary without any manual intervention. Similarly, derived state is a solved problem: if it goes out of date, it needs only to be regenerated. The Unix make utility is already routinely used for this.

Manual config state is a tremendous source of futz in most systems today. Worse, it is the most difficult kind of futz possible: unlike policy state, where various alternatives work but may not be desired, most of the questions answered by manual config state have only one or two right answers and plenty of wrong answers, and wrong answers generally render the system or components of it completely inoperative. Ultimately, this is the category of futz that is most seriously in need of reduction. Fortunately, it is possible to accomplish this: to the extent that there are right answers, in almost all cases, with sufficient engineering of components, those right answers can be probed or determined from context. For instance, the only reason we need video card and monitor information in /etc/XF86Config is that on PC-based systems it is not possible in many cases to safely or reliably interrogate the hardware to find out what it is. In a hypothetical world where you could query this hardware, which is easy to imagine, this major source of futz could be abolished.

OS file state and application file state are an area in which many current systems fall down: it is quite easy, in general, to install new application software that breaks the system, or to update the system and thereby break applications. It is also possible to delete or rename important files inadvertently (or lose in a power failure) thereby breaking the system. At present, recovering from these problems is generally quite difficult. In this area, for most people, futzing at all tends to equate to reinstallation.

Reducing this category of futz requires taking more care in analyzing the dependencies among software components, and improving the mechanisms with which software components are bound to one another at runtime. We need several things: automated analysis of runtime dependencies (a hard problem), better systems for preventing accidental version skew, and mechanisms for cross-checking that can be performed at runtime to allow failures to occur gracefully. Reinstallation as a failure recovery mechanism is unacceptable.

User file state is inevitably a source of futz as things become disorganized and users mislay their data. We see no immediate prospects of cutting back on the futzing this requires, although developing a good model for how applications should choose default save directories and the like would be a good start. Content indexing techniques may be of help as well.

Application context is normally automatically maintained, and only becomes a source of futz when it becomes corrupted or saves an undesired application state. This problem is easily solved: check it for consistency when loaded, be able to withstand it being deleted, and store it in a known location so users can delete it if they so desire. In many cases, simply not keeping such context is an adequate solution.

System context is essentially the same, except that it is sometimes not possible (or meaningful) to erase it and start over. It is much more important to check it for consistency and repair any problems. With some engineering, failures that require expert attention to repair can be made quite rare, as they generally are with most Unix implementations of fsck.

Cryptographic keys are listed separately because they have their own unique requirements for management, and because they are mandatory for the use of secure autoconfiguration protocols. In our experience, these are not large sources of futz. Furthermore, a lot of attention has already been paid to key management in the security literature.

All the above assumes that a user is changing state in order to make some kind of desired configuration change, either as ongoing maintenance or at system installation time. There are two other cases in which one needs to interact in intimate detail with the state of a system: to diagnose and

repair a system failure and to monitor the system for signs of upcoming failure.

Properly speaking, as we have defined futzing, diagnosis is not futzing; rather than experimentally adjusting state to achieve a result, diagnosis properly involves analyzing existing state. Sometimes, however, one needs to experiment to interpret the existing state. And additionally, a common method for recovering from a system failure is to futz until the obvious signs of the failure have disappeared and the system appears to be working again. (Rebooting is a drastic example of this technique, and it works because much system state is not persistent across reboot.)

The reason this method works is that many system problems involve the failure of supposedly automatic state management mechanisms; tweaking the state tickles the state management mechanism, and with some luck it will start functioning again. The reason it is common is that actual diagnosis by inspection usually amounts to debugging and requires an extremely high level of expertise.

If the system can diagnose problems itself, like our copier can, this futzing becomes unnecessary. Even if it can only diagnose a small number of the most common problems, a good deal of mandatory futzing can be eliminated. Self-diagnosis in software systems is an important research area. We believe a good deal of progress is possible.

Monitoring for signs of upcoming failure, including monitoring for security problems, does not, itself, involve futzing. However, failure to perform monitoring can lead to huge amounts of futzing later on - recovering from a server dying can easily take as much futzing as installing a new one, whether the death took place because of hardware failure or because of hackers. Therefore, automatic monitoring is also crucial to building true no-futz systems. This is another important research area.

Ultimately, all of these things - monitoring, diagnosis, and configuration - involve interaction with the system state. We believe that research and engineering in the areas outlined above can tame a good proportion of the typical system state space. However, policy state, cryptographic keys, and probably some leftover bits of state in the other categories, are not going away. More is required; we need to be able to manage this state.

4 Futz and State Management

The less state a system has, the easier it is to organize and present to users in a coherent manner.

As outlined in the previous section, one can design out some state and automate the handling of a lot more. This will take care of a good deal of futz. However, a great deal of state remains, and it requires editing, and undoubtedly, futzing. One cannot eliminate the editing. But one may be able to eliminate the futzing.

The leftover state consists mostly of policy state, manual config state, and autoconfig state. This state can be thought of as a list of configuration questions and their answers. The ultimate goal is to allow a user to type in answers to these questions, or change the answers to suit changed circumstances, without needing much training or specialized knowledge.

It should now be clear that question formulation is crucial — not just their wording, although that is significant, but what questions are asked, how interconnected they are with each other, how they're grouped, etc.

What this means is that, once all the easier issues are addressed, the organization of the state space of the system is the most significant factor determining how much futzing the system will demand.

It is crucial to analyze this state space in detail and determine how to best decompose it into a set of variables (and thus questions). In the best such decomposition, the variables will be as simple and as orthogonal to each other as possible. It will be clear what answering each question entails and who, in any of several typical environments, ought to decide the answer. Then the questions need to be written in such a manner that the people who typically fill these roles can, in fact, answer the questions without needing an excessive amount of training, and the software needs to be written so that questions will not be posed to the wrong people.

For example, in almost all cases, the person sitting at the computer should be the one to choose the desktop background. However, it is not necessarily the case that this person should be asked "What is the IP address of your web proxy?" — this question may need to be posed, but if so it should be posed in a context where it is clear that the answer is the local network administrator's responsibility.

We believe this is the key. It is not an easy problem; in the absence of any useful decomposition theorems for state spaces or state machines, it must be solved by manual inspection and ad-hoc heuristic analysis. Worse, one has to address the complete state space of the entire system at once; if one leaves some state out of the analysis and tacks it on later, it is almost guaranteed to be a poor fit.

At first glance this might seem to mean that all application software must be designed into the system. This is not the case. However, what *is* necessary is for the sorts of state applications may need to use to be anticipated; that is, one needs an abstract model of what an application is and does. Such a model should be reasonably general without going overboard: applications that fail to fit will still work, but may require increased amounts of futzing. Allowing for these applications in the general design might result in even more futzing in the common case. There will be a trade-off, and that trade-off will need to be explored.

5 Research Directions in No-Futz Computing

If the systems community is to ever build no-futz systems, we must embark on a research program that addresses the key issues in no-futz computing. This section defines those areas.

The first step on the path to no-futz computing is determining how to measure a system's futz. We wholeheartedly

endorse the term "FutzMark" coined at the last HotOS and challenge researchers to define it.

We believe the central issue in no-futz computing is state management. We must reduce system state to a manageable level, isolate each state variable so that it is orthogonal to other state variables, and make it impossible to specify invalid states. Where possible, we should replace state with dynamic discovery. Where possible, we should devise ways to turn static state into dynamically discoverable state (e.g., autoconf data, manual config state). Achieving orthogonality is perhaps the most difficult aspect of this task, but also the most essential. Without orthogonality, the problems of management and testing grow factorially. If we can achieve orthogonality, it becomes a manageable linear problem.

In lieu of total orthogonality, we need better mechanisms to identify inconsistent state and remedy it. We need to identify (or avoid) version skew among software components and do more extensive runtime cross checking and analysis.

Coping with failure requires a great deal of futzing; thus we need to achieve cleaner failure models. In the fault-tolerance community, "failstop" behavior (ceasing operation as soon as a fault occurs) is considered desirable so that failing systems do not corrupt state or data. In the context of no-futz, failstop behavior could permit the precise identification of failure causes. If systems can diagnose their own failures, it's conceivable that they can then direct users to perform recovery, as our copier does. In general, we need to make progress in the areas of self-diagnosis and automatic monitoring.

Finally, there are areas outside of systems research where progress is necessary. In particular, improvements in user interfaces and data presentation will reduce futz. Collaborative interfaces, which act as intermediaries between users and their machines that enable them to work together, hold great potential if applied to no-futz computing. Security management is sometimes considered outside the realm of systems, but insecurity is a major contributor to current futz and improvement is needed. Improvements in content indexing will reduce the futz associated with user data management.

6 Related Work

There have been a number of efforts to reduce futz in computer systems. In a distributed setting, Sun's Sunray [4], as well as Microsoft's Zero Administration initiative and the associated IntelliMirror [7] product, are projects to centralize futzing.

The Sunray system's desktop machines are simple, stateless I/O devices with no administration needs. Sunray relies on modern off-the-shelf interconnection technology and a simple display update protocol (SLIM) to support good interactive performance. In addition to eliminating client administration, the Sunray model offers client mobility. Client session state is entirely stored on the server and can be associated with a smart card that can be inserted in any Sunray client connected to the same server. Sunrays are anonymous commodities. However, this does not eliminate the administration cost. Sunray servers are complicated systems and not easy to administer: once, in our department, one of the junior system administrators broke all the Sunrays for three days just by trying to install a new utility on the Sunray server.

Microsoft's Zero Administration initiative is an effort to reduce the administration needs of Windows installations and thus the cost of ownership. Central to Zero Administration is the IntelliMirror product, which helps an administrator (a) manage user data, (b) install and maintain software throughout an organization, and (c) manage user settings. Management of user data requires knowledge of properties and locations of users' files so that the data is available both online and offline from any computer. Manual installation, configuration, upgrades, repair and removal of software across an organization requires large management effort. IntelliMirror automates this: it offers remote OS installation, a service allowing a computer connected on a LAN to request installation of a fresh copy of the Windows OS, appropriately configured with applications for that user and that computer.

Sun's Jini [5] for Java is an example of a system that tries to eliminate administration in a decentralized ("federated") manner. Jini provides a distributed infrastructure for services to register with the network and clients to find and use them.

7 Conclusion

Leading systems researchers identified no-futz computing as an important research area two years ago [3], but to the best of our knowledge, there has been no significant research activity in this area. We believe one reason is that the problem is enormously complex and may not be solvable within the constraints of legacy systems. Regardless, until we identify the important research questions, no progress can be made. In this paper, we have identified some, if not all, of the important areas in which research must be conducted if we are ever to "solve" the problem of high-futz systems.

8 References

[1] Forrest, S., Somayaji, A., and Ackley, D., "Building diverse computer systems," *In Sixth Workshop on Hot Topics in Operating Systems*, 1997.

[2] Dan Plastina, "Microsoft Zero Administration Windows", invited talk given at the 11th USENIX Systems Administration Conference (LISA '97), October 26-31, 1997, San Diego, California, USA

[3] Satyanarayanan, M., "Digest of Proceedings", Seventh IEEE Workshop on Hot Topics in Operating Systems, March 29-30 1999, Rio Rico, AZ, USA.

[4] Schmidt, B. et al., "The interactive performance of SLIM: a stateless, thin-client architecture", in Proceedings of the 17th SOSP, December 1999, Kiawah Island, SC, USA.

[5] Waldo, J., "The Jini Architecture for Network-centric Computing" *Communications of the ACM*, pp 76-82, July 1999.

[6] Conversation with Xerox Technical Representative. January 18, 2001.

[7] http://www.microsoft.com/WINDOWS2000/library/howitworks/management/intellimirror.asp as of April 23, 2001.