# Proceedings of the 6th International Workshop on Plan 9


**October 20— 21, 2011**
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain

*Dennis Ritchie, in memoriam.*

*Without his brilliant work, none of this would have been possible.*

*He is still an inspiration to all of us.*

**Organization**

Francisco Ballesteros, *Lsub, URJC*
Gorka Guardiola, *Lsub, URJC*
Sergio de Mingo, *Lsub, URJC*
Erik Quanstrom, *IWP9.org*
Enrique Soriano, *Lsub, URJC*

**Program Committee**

Sape Mullender, *Bell Labs* (PC chair)
Francisco Ballasteros, *Lsub, URJC*
Brantley Coile, *Coraid Inc.*
Charles Forsyth, *Vita Nuova, Ltd.*
Jeff Napper, *Bell Labs*

**Participants**

| | |
|---|---|
| Francisco J. Ballesteros | nemo@lsub.org |
| Federico G. Benavento | benavento@gmail.com |
| Axel Belinfante | Axel.Belinfante@cs.utwente.nl |
| Nicolas Bercher | nbercher@yahoo.fr |
| Ernesto A. Celis de la Fuente | ecelis@sdf.org |
| Brantley Coile | brantley@coraid.com |
| David du Colombier | djc@9grid.fr |
| Andrés Domínguez | andresdju@gmail.com |
| Vincent Douzal | vincent.douzal@teledetection.fr |
| Bruce Ellis | bruce.ellis@gmail.com |
| Brett Estrade | estrabd@gmail.com |
| Noah Evans | noah.evans@gmail.com |
| Charles Forsyth | charles.forsyth@gmail.com |
| Jesús Galán López | yiyu.jgl@gmail.com |
| Nick Gawronski | nick@nickgawronski.com |
| Gorka Guardiola | paurea@lsub.org |
| Pedro Hernandez Jr. | mesosnak@sdf.lonestar.org |
| Latchesar Ionkov | lionkov@lanl.gov |
| Lucas | nexusst@sdf.lonestar.org |
| Stephen Jones | smj@sdf.org |
| Eric Jul | eric@cs.bell-labs.com |
| Matt Lawless | maht@maht0x0r.net |
| Cinap Lenrek | cinap_lenrek@gmx.de |
| Christoph Lohmann | 20h@r-36.net |
| Mathieu Lonjaret | mathieu.lonjaret@gmail.com |
| Hugo Marcelo Rivera | hugo.rivera@unimi.it |
| Richard Miller | miller@hamnavoe.com |
| Sergio de Mingo Gil | sdemingo@lsub.org |
| Christian Mongeau | chr+plan9@tx0.org |
| Sape Mullender | sape.mullender@alcatel-lucent.com |
| Jeff Napper | jeff@plan9.bell-labs.com |
| Erik Quanstrom | quanstro@quanstro.net |
| Jan Sacha | jsacha@plan9.bell-labs.com |
| Nima Sahranshin Samani | unix.nima@gmail.com |
| Henning Schild | henning.schild@plan9.bell-labs.com |
| James Schneider | jim2161@att.net |
| Brendon Schumacker | brendon.schu@gmail.com |
| Frédéric Da Silva | frederic.da-silva@9grid.fr |
| Steve Simon | steve@quintile.net |
| Enrique Soriano | esoriano@lsub.org |
| Ataliba Teixeira | cerebro@freeshell.org |
| Roy Williams | rwilli95@uncc.edu |
| Jeff Woodall | jgw@sdf.org |

# Foreword

This book contains the proceedings for the Sixth International Workshop on Plan 9, IWP9. It was held on the 20th and 21st of October of 2011 at ETSIT, Rey Juan Carlos University. We, the organizing committee are proud of hosting this workshop again. Back in 2005 when we hosted the first one, we did not imagine it would continue for so long, getting to be the meeting point for the Plan 9 and Inferno communities.

The workshop includes a tutorial by Sape Mullender on the important topic of performance evaluation. There is also a panel discussing news from new systems and ports, new sprouts coming out of Plan 9 an Inferno: nix, osprey and new ports of inferno. A lot of work has been going recently into these new systems and furthering the reach of the Plan 9 approach.

This workshop was organized by the Systems Lab, a group in the Systems and Communications Group (GSYC, URJC), and Erik Quanstrom. It would have not been possible without the financial and logistical support[1] from Rey Juan Carlos University and the Comunidad Autónoma de Madrid, which we would like to thank as well.

We would like to thank Sape Mullender, Brantley Coile, Charles Forsyth, and Jeff Napper for being in the Program Committee. We also thank the original writers of Plan 9 and Inferno for their insight and the great job they did designing and programming this wonderful system and the community of people around the world who keep contributing to it.

Proceeding updates and an online version of this book are available at the website http://iwp9.org.

The Organizing Committee:

Francisco J. Ballesteros
Gorka Guardiola
Sergio de Mingo
Erik Quanstrom
Enrique Soriano

---

[1] This workshop is supported in part by Spanish CAM grant S2009/TIC-1692

# Table of Contents

# Tutorials, Panels, and Workshops

- Tutorial: *Measuring Performance*. Sape Mullender.
- Panel: *News from new systems: osprey, inferno-ports, and nix*. Sape Mullender, Charles Forsyth, Francisco Ballesteros
- Workshop: *Hellaphone*. John Floren
- Workshop: *Jtag*. Gorka Guardiola

# From natural hazards to outer space and to Plan 9

*Vincent Douzal, Cemagref, UMR Tétis, Montpellier*
vincent.douzal@teledetection.fr
*Nicolas Bercher*
nbercher@yahoo.fr
*David du Colombier*
djc@9grid.fr
build date: 2011-10-13 22:30

*ABSTRACT*

An information system for scholarly work on natural hazards calls for the design of a computer system for transmission of information over very long periods, and for traceability. An abstract analysis shows that these requirements are dual to the fundamental question of assisting the cognitive activity of a user using external memories, which reaches a very general scope. The solutions should be implemented at the operating system level, mainly the file system, and Plan 9's file systems and other properties make it the soundest base for our work. We present our road-map for development.

## 1. Introduction: the case of a computer system for scholarship on natural hazards

We were requested to design an information system supporting scholarly work on natural hazards. It quickly appeared that all visible, mainstream systems sorely lacked some core properties, thus incurably disqualifying them, notably in their relation to time. We therefore considered a more fundamental problem: building the necessary foundations of a system to work on natural hazards. Early on, it became obvious that many requirements for the system were not specific to that field. Even so, 'natural hazards' provides a good concrete example to guide the design. For example, it naturally involves century-long durations, which has significant implications for the design. Furthermore, by their nature, the natural events that will form the data records are unpredictable, and their nature often precludes making precise observations at the time. Crucial data arrives unformatted from endangered persons. Later, historians will delve into archives, sifting centuries-old files, seeking to reconstruct events. That is exactly the process we want to assist. The context of the reconstruction or investigation matters too. For example, if the work is contractual it might be brought before a judge for arbitration, which would proceed by retracing its sources; similarly, a scientist or scholar might need to retrace arguments made earlier by others, and check results. On behalf of the earlier author of any conclusion, one should be able, provably, to revert to the exact state of information he had recorded at a given date, and perform again the same processing steps. Thus, data collected today might still be required in a remote future. For working on natural hazards we see a need to design an information system for the next 500 years! That leads us to the question: what is invariant over such time scales?

The ''sciences of traces'' — history, archaeology, paleontology, geology, cosmology (on ''memories of the world about itself''), the now fashionable forensic medicine — all share the essential structure of a police investigation. This hints that, when thinking on scales of centuries, any design will be totally unspecific to any particular application domain. Traceability is the key element to reproducibility in science. Let us establish that two questions

are dual in the mathematical sense (they are two sides of the same coin): 1. The very long-term transmission of knowledge established on digital memories. 2. The absolutely minimal functionalities a computer should fundamentally provide for sustaining, assisting a user in his cognitive activity.

All the usual assumptions for designing an information system are swept aside by duration. Space coordinates are taken as an invariant index in geographical information systems, but space shrinks or expands during seismic events. Continents drift. Concepts evolve. Languages change. No application ''ontology'' could be expected to last, only very abstract concepts. One cannot envisage a data model (in the usual sense) for a very long-term repository. Hence a very long-term database should be completely agnostic to the kind of data it will store, and have no data model at all.

## 2. Into outer space: An abstract analysis, for the essentials of the structure of an external memory system

Our model of the system can be expressed visually by Figure 1. The α part depicts our expanding universe unfolded in space-time, in a way standard in physics since Minkowski. The world at a given point in time τ is a slice of space-time, represented as a Venn diagram: the set of perceptible events at τ. By construction, events are fixed points: unchanging, invariant entities. The arrow of time τ is irreversible.

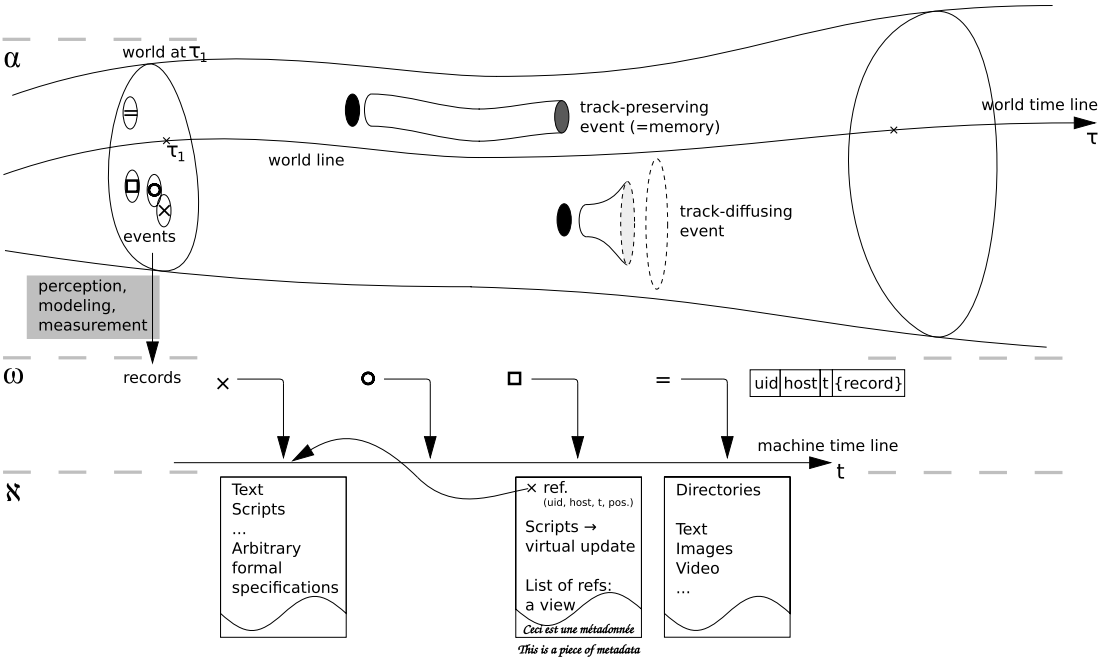



*Figure 1. The whole article can be considered an extended legend to this diagram for the abstract setting of our design. The idea of including α in the form of a Minkowskian diagram was inspired by Schueler's remarkable paper[17].*

Two notable kinds of events are shown: those leaving long-lasting tracks (a fossil, a footprint) appear like tubes, and those which don't (like a perfume spray). Observing a slice of

a tube implies an event in retrospect, if only one has the ability to interpret tracks, and traces. Otherwise past events are lost.

If we are to capture facts about events (in ω), we need to take their image under a given relation, in the sense of set theory. An image in the form of a series of symbols is the convenient way to put things in writing, into an external memory. This relation from events to categories and eventually symbols takes varied forms, some are termed perception, modelling, measurement, or even database form filling. Recordings of these assignments come in sequence along a time line.

The α and ω parts display an inviting symmetry. Since events are fixed things, so should be their images: written once-for-all, never altered again. Images, if they are to play their rôle, must be written on lasting, immutable external memories—tubes. External memories act as messages; they offer to parties of a contract the cross-checking abilities that are essential to their use, like the physical tokens of Neolithic accounting systems [16].

The whole purpose of the diagram's ω is to break symmetry with α on time; unlike τ, t is reversible: it can be travelled up and down. The impossible motion back and forth on τ is replaced by a motion in space, between preserving tubes of memory indexed by t.

Together with *t*, a user ID *uid*, and a unique repository name *host*, form the triplet ID of a record. These are the only metadata elements guaranteed to appear for every record. (Obviously, they are strictly internal to the recording system.)

Note that the machine representation for t does not have to start at the (hypothetical) origin of τ for the universe: an artificial epoch is enough, such as 1970 in Unix time, or the more recent date of establishment of the repository. The stated dates of events are to be written in the records themselves, using for τ a calendar representation, to which machine time must naturally be connected (with that connection deposited periodically in records). Pointing to cosmic, geological, paleontological remote events on τ is done by ''inductive referral'', through a theoretical construct. For example, consider the chain of hypotheses and processes involved in carbon-14 dating an artifact as 14,000±500 years before present, which in effect builds a virtual time arrow. Similarly, description of where events take place is not primary, but secondary; it is of course of importance, but cannot be enforced in the records.

Machine time t is the only organising dimension of the repository. It is a simple order, expressing the intrinsic 'sequentiality' of any subjective experience. Sequentiality is the only structure inherent to the system by design. It is the only indispensable and unavoidable structure that should possess an external memory system, as reflected by the diagram; and no other structure should be put in without harm.

An elementary cognitive operation consists in picking two arbitrary memories, experienced at any times, and relating them in some manner. Thereby for instance an insignificant event in childhood suddenly makes sense in the light of a recent reading. Relating two events is done in א by referring to their records. In an א record, referring to a past record is to open a *view* on it: a generalisation of what a directory provides on usual file systems. A view can gather arbitrary records, comment them freely, support action on (parts of) its constituent files through scripts, and include visualisation and user interaction facilities. A view could also be used for instance to provide another user with visibility and rights of accessing a given set of files, or to display several parts of files as a single body. We see here the demand for a computer language to express such views.

Diagrams ω and א represent well the progress of, say, an historian's trajectory through an archival corpus, making records, reconsidering them, viewing them arranged in any order he likes. More broadly, it represents an investigation in any ''science of traces'', and even more generally, the acquisition of knowledge along a lifeline, in a linear succession of encounters of people, situations, books, etc., all marking events in time. At an abstract level all these processes are formally identical: ω and א capture both some essentials of cognitive activity and of an external memory system for expanding, augmenting cognition.

A ship's log book is a perfect characterisation of a lifeline, it reflects in its sequential records the ship's navigated trajectory around the sea, and its conjunction with significant events at each time and place. The corresponding basic information system architecture is that of a computerised log book.

Our design minimally implies two components, a linear, sequential file system, strictly 'write once', and a language to express free comments, references to records, and invoke arbitrary scripts on records.

## 3. Long-haul message passing into the future: the need for a corpus

Any writing is a message sent into the future (if only to oneself). A message makes sense because of a convention, a contract between parties. If my collaborator voices ''42'' over an experimental equipment, that can be a perfectly qualified piece of data between us. On the other hand an isolated file containing ''42'' makes sense to no one. Agreeing on a given protocol allows a message to achieve its intended purpose, carrying meaning, and is called interoperability. Applications agreeing on a file format achieve so-called technical interoperability. Conforming to shared ontologies allows so-called semantic interoperability. By contrast, messages between humans have no static semantics; they must be submitted to interpretation, just as any track requires the skill to interpret it to reveal a past event. Interpretation needs to know the surrounding context of the message, a larger context with cultural distance.

For data, that context is the *corpus*: a substantial body of information with internal closure and additional connections to our present world, enough that we can interpret a given item. The further into the future a message is sent, the more it needs a whole corpus of documents as a context to convey sense, because interpreting the message implies reconstruction of the whole sender-receiver-transaction-motivation setting. Protocols and conventions, and cultural values evolve. Scripts get forgotten, and only when a substantial corpus of connections, with enough closure is available can they be deciphered, as with Egyptians hieroglyphs or the Maya script. It follows that a corpus is the best basis of trust. Internal closure eases forgery detection. Traceability is a fair basis for reputation.

The DOI system documentation [3] asserts that managing data implies managing relationships between entities: A has the relationship B to C. For instance, ''Albert Camus is the writer of La Chute'', an expression that directly maps into, say, RDF. Provenance should not be omitted: who said that, and when. Relationships between entities is often called metadata.

A corpus is ''extreme metadata''. Clearly any record in our repository is metadata to any other one it references, and conversely. Even more so, any set of contemporary records are implicitly but strongly connected just by being close in time: a Freudian slip, or even an empty record, can convey meaning just from its position after an otherwise unrelated record.

## 4. Contrasting with current architectures: enforced order

All current systems force data into a Procrustean bed of arbitrary orders, ultimately distorting it to the point of defeating its use, and even obstructing its entering the data system.

Current file systems insist we arrange files in a hierarchy. Thus we try to express semantics in a hierarchy. How long can a hierarchy, however carefully devised, accept new subjects without breaking? Also, much too often, a file should legitimately be in two different places. Large libraries confront the same dilemma [18], when two yet unrelated disciplines, remote in classification and physical arrangement give birth to a completely new one. The negative consequences of imposing arbitrary order on data where it does not belong poisonously pervade every task. A file system hierarchy also cannot grow indefinitely without requiring foreign mechanisms, such as logical volumes. By contrast, our linear sequence of records naturally splits up in articles and extends easily by adding memory

devices. For the same reasons, it lends itself well to parallelism.

Relational databases similarly force data into the relations of a given data model. Diagram ω shows how a very long-term database can be agnostic to the kind of data it will store, by reducing to a single table with key *uid, host, t*. Hence no database system is needed, just a file system. (If you do want a database, however, there is no difficulty in periodically storing its current state and log inside a record.) A database model somewhat supposes it exhausts the possibilities of what will have to be recorded in the future.

Forcing inappropriate order, current systems make it difficult to accommodate really new concepts. What protocols, what conventions will be demanded in the future, we cannot foresee and must not prejudge.

The computer system outlined in the diagram — a computer being a memory with an engine to process data — retains only the minimum necessary to support cognitive activity, and allow transmission of knowledge into the distant future.

### 5. Storing files as the privileged digital abstraction

What digital abstraction should we make accessible to the user? Files are almost universally available, well understood, and uniformly usable: they hold everything. We feel we also cannot avoid file hierarchies, because most systems present themselves in that form and can only be used in that way. Accepting hierarchies does *not* mean forcing everything into hierarchies: it is only a facility offered locally in a record.

The intended linear, sequential file system uses Venti [10] as block storage layer. Unicode is the encoding of choice, with its universal scope and strict stability principle, consistent with our concern.

### 6. Correcting errors without hard update, and the generality of references

Requiring traceability has a drastic consequence: since updating contradicts traceability, in our system you can neither erase nor change any data, once recorded. Figure 1 illustrates that traceability is an integral part of cognition, inseparable from it, and that in this design a correction is impossible, because the past cannot be changed. Similarly, a hard update operation, by rewriting data in place, would contradict the essential function of a memory system. Instead, one may specify a view with arbitrary transformations to apply to any previous record. Given this mechanism, most operations become surprisingly simple, because everything written is immutable; thus, if something works today, it will work forever. (Suppose that necessary hardware architectures can be emulated or virtualised.) There is no potential for 'inconsistent updates' when updating-in-place is avoided: layers and layers of comments may overlay one another, even contradicting each other, but without inconsistency.

Even more important, time t is reversible, by building suitable views. Plan 9 is a living demonstration of the soundness of suppressing update, at a certain level, in its main file system, allowing the *yesterday* command to look into the system's past. In his ''Debugging backwards in time'', Bil Lewis [8] also shows the power of reversible t: by recording every state change in the run of a program, you can navigate the unfolding of every bit of information that might be useful (just as all events in α cover all the possible sources of information, for ever).

### 7. Letting the user unobtrusively express his own order and trace it at desired detail

A single reference mechanism, at the level of records, accounts for citation, quotation, annotation, comment, or extracting data to feed into a process.

All the expressed order, all the structure is formed within records (especially as views), except for the unavoidable temporal sequentiality. It is the only place for variation, and it gives complete freedom: arbitrary structures can be expressed. Current systems tend to

mix places where order is expressed, and maintain a confusion between stating ''In my present opinion, this piece of information is incorrect, because such and such'', and the urge to rewrite the past. This approach is rather different from much work on file systems with semantic concern [5], though not necessarily incompatible with it. Reference and annotation, not updating in place, is the general case, and appropriate to traceability. The essential thing is to let the user specify his own order, rather than impose an arbitary one, and to ensure traceability in a non obtrusive way.

In cognition, selected events are recorded, and structured into one's own order. Some events are left dormant, some are continuously reactivated and reused, but few if any are totally forgotten. Cognitive psychology and perception theories consistently see the world as a chaos — in the sense that its structure is foreign — that each subject is challenged to organise, to render consistent for himself. (The chaos was represented in Figure 1 by an amorphous set in $\alpha$.) When represented in an external memory, that organisation is also personal.

The very nature of a personal work is to impose one's own order on things. Generations of historians can follow one another working on the same corpus without repeating themselves; in this type of scholarship the essential contribution of each is offering his own model to interpret data, rather than simply applying a borrowed model. For example, many scholars reorganise their bookshelves for their current project, and workers often keep a selection of books or papers at hand, on a desk.

The decision of what to retain is a recurring one, and all too easy to overlook. For example, you receive a Sibylline email with a URL ''explaining it all''. The email was archived, but years later it no longer makes sense since your sender forgot to save the world simultaneously, and the URL vanished. If a reference is really useful, its name alone can do nothing: something must be retained of its content.

So far we have treated the architecture displayed in the diagram as entirely devoted to a personal repository. But since the structure with its triplet ID is absolutely minimal, it is not surprising that it extends smoothly to a collective use. There is no operational gap between personal and collective work in this architecture. The result of contemporary or delayed annotations is a fairly thorough record of the cognitive interaction of users. Users can use the same repository independently without harm, if they wish. There is in principle no obstacle to a very large number of users. Exporting the trace of a record is easy, by capturing all the cascade of dependencies (which can be very large if a view says, for instance, that it discarded very many records for reasons it extensively exposes as justifications). Records from different repositories can easily be mixed and disentangled. The *write once,* sequential structure makes it easier to do distributed replication. Backlinks are easy to find with an index, much as the web is indexed for search engines. The index can grow incrementally, and be recorded just as any other data.

Having all the structure expressed in records has an important consequence on the use of names. Names given at the birth of an object are important, for files as well as for variable names in programs. But new names are called for when going into foreign languages (or staying in the same place, waiting long enough). References and views allow naming, assigning names appropriate to the current use for a file, a set of files or portion of a file, or a set of portions… any describable structure. Views open the possibility of a unifying namespace that could stand for centuries, evolving but always ultimately referring to the same original collection of data. Just as we have seen that metadata (relating two records) is by nature always an afterthought, appropriate names, which are just a particular type of metadata, are often found belatedly. A new way to *view* things is often a new way to name them. Certain names come to be accepted as labels, which restrains both the proliferation of names and the complexity of their relationships. A synthetic view can also cut back complexity.

## 8. More on personal vs. central repositories

There is more to justify personal repositories. ''Information empires'', where data stores are more and more concentrated, cannot be blindly relied on as external memories, however self-confident their security, or however open their systems. Any technical organisation can break. Natural hazards lurk. Empires by definition are subject to radical decisions. It is reported that Qin Shi Huang, the first emperor of China, had all existing books not complying with Qin historians burned (with exceptions for a few special fields), and later buried alive scholars who either still owned those books or still had them in mind and could spread their ideas. The library at Alexandria was destroyed. Information empires today are not in a different position. In mid April 2011, Google announced that on 29 April, 2011, files hosted on Google Video would no longer play, and completely shut down the service on 13 May [1], thus disproving those who promoted video sharing as an archival solution. Users, if still alive, had that short time to hear about the shutdown, and quickly download their files again, because no archives were to be kept.

In the same vein, many initiatives of data preservation are focused on large data sets [7]. Among many reasons, large projects make it easier to attract funds, and to impose on their users constraining technical choices to ease administration. But the problem is actually to capture people's work the way they work: that is, how they interact with data. Any one seeking today for a long-term digital preservation solution on a personal basis is left without a solution, even without mentioning the inability really to address the concern for preservation in everyday activity. But preservation, archiving, cannot be an afterthought, all the more so in the digital world. The history of science teaches us that the most surprising breakthroughs come from the most unexpected conditions, and they are by nature impossible to predict. If we want to capture them, we need a solution to hand. Meanwhile, a huge amount of knowledge is continuously being built by hosts of individual researchers around many small, valuable data sets. Then some day scientists retire, and the traces of their knowledge retire with them. And it is not uncommon that a few month after submitting a paper, traces of how to reproduce the results are lost [2].

In the scientific field, we witness in practise that the more computers become intimate at every stage of a scientist's work, the less a given piece of work becomes reproducible, whereas we might expect the exact contrary. The creeping of instant messaging into scientific work, and the occasional vanishing of data on the Web, challenge more and more one's ability to capture them efficiently, and to retain appropriate images of them to make them usable as reliable references.

By reading a novel, one can experience someone else's slice of life, without having to actually live it. By borrowing someone else's library, one can watch through his cultural window. By borrowing someone else's repository, one can go even further, with the choice of jumping to conclusions, and backtracking freely to the sources, nearly walking by the stream of thoughts of a forerunner. The projected system, serving as a general digital laboratory notebook, would give that incredible ability to follow in someone's footsteps. If we want to ''climb on the shoulders of giants'', we must be able first to follow in their footsteps (in their tracks). We argue that we certainly need solid, provable *personal* digital memory systems that are easy to manage.

## 9. Implementing a linear (sequential), write once file system

This section is even more a call for comments than the rest of this paper. It should now be obvious that an adaptation of Fossil [11] reaches the specification we aim at.

There will be, side by side, a user's workspace (home directory) together with the set of files necessary to boot the operating system into a particular, functional state, and a specific deposit area in which to put files or directories that will become a repository record by a sort of *on-trigger* prompted snapshot process that builds a specific Vac file with the attached triplet (*uid, host, t*). (A typical t would be an integer count from an arbitrary

origin, convertible in calendar time; note that for instance 64 bit at a nanosecond resolution represents about 584 years.) Venti need not be changed.

It is up to the user to determine the chunks of data (e.g., a whole directory or many separate files) that are to become records; the main rationale is his own convenience. The semantics of 'gathering files' initially is not particularly strong and is likely to be split into more specific sets in the future.

Eventually all the files are found in the repository. It seems that the notion of a home directory is necessary to hold at least a list of maintained views on what my work has been so far, or what my workplace now is. But in fact, a single view is enough, and perhaps experience will show that the distinction of a home is unnecessary and it suffices to fetch the last personal view from the repository. A view might also allow fetching all the bindings needed to boot a particular operating system state and file system configuration. As a result, only a transient workspace where current work takes place might be enough. Caching that workspace, to have it at hand immediately when one returns to the machine, is close to having a home directory.

Of course some bootstrap is required to first explore enough of the repository to be able to load and jump into another machine. Having the full hardware-software stack available is the definitive—the *best possible*—condition for allowing a user to interact with a given file fully. For preservation purposes, confining the main work inside a virtual machine allows saving and restoring that complete hardware-software stack. Most of the work should unfold on the repository, but determining how much so is reasonable, still needs to be determined by experiment.

The sequential record structure means that cutting a large repository into stripes as it grows to huge sizes is easy. Parallel access is then possible, which conversely results in duplicating some blocks, with an independent Venti for each stripe.

We assume it is or will be possible to monitor media memories for degradation (which is eased on a Venti-based storage system), and migrate them accordingly, thus ensuring continued access to the bits of a file. Many copies also make data safer [13]. Well-traced, workable, computable corpora as planned here will undoubtedly be better candidates for the effort.

## 10. A language for the elementary interaction and its text editor

References—(hyper)links—are expressed as text, pointing to a record by a triplet uid, host, t, and a position. In a text file, a position is a simple integer (if you count in characters, different from bytes in UTF-8), but segments of text can be located by any other mean, like a string search, a regexp, or using the patterns and region intervals of ''lightweight structured text processing'' [9]. (Remember that if a string search leads unambiguously to a position, it will always, since files don't change.) In an image or other special formats, appropriate locators must be used.

As a minimal effort implementation we use Emacs' Org mode from a Linux system. Org is a general purpose tool and a coordinated light syntax (in plain text files) for keeping notes, outlining text, internal and external links, including source code blocks and evaluating them, and converting them into source files using the noweb convention of literate programming [6]. Exports to various formats including LaTeX are easy, it also comes with many useful facilities as timestamping and calendrical calculations (so useful to historians), agenda views, tagging, maintaining to-do lists, very fit for project planning and perfect for keeping a log book, and at the moment, to give a feeling of what could be the work on a log book structure.

The aim at this stage is to do the work mainly from a Linux installation, with great use of Emacs (we know we are in a state of sin, but it compensates for many gaps of integration between the operating system and its windows). All the repository is kept on file systems under Plan 9. Nearly all the facilities we need are present under Emacs, or otherwise

relatively easy to develop. Commands like find-dired or find-grep-dired in Emacs can give users a good intuition of what a view might be, in general, and of the power of text manipulation. There are, however, exciting prospects in a combination of features from Acme and Emacs, under the Plan 9 namespace.

Linux is not too exotic for a fully-fledged environment for a large set of potential users, and is becoming Plan 9 tamed. It is realistic not to expect people to shift from their habits, especially if we look more widely for future users, especially in the humanities or other fields ''foreign'' to computer science, to experiment with the system. One of the ultimate purposes is to be able to tempt users of any operating system with a facility for traceability that mildly changes their habits, though inevitably some systems will offer a result with rougher edges than others.

This combination we aim at offers the least-effort path to a minimal working installation of what is meant as a general purpose environment, as seamless as possible from programming to common use. It is difficult and useless to try to foresee what usage patterns will emerge, and nothing can replace hands-on experience. For instance, whether there should be some sort of reserved keyword to state ''This is a correction'' is left open, though intuition suggests that we ought to avoid it: proper computer semantics can reside in the repository's scripts, and fine human semantics can be written as subtly as poetry, in an accompanying comment. And natural languages evolve... Leaving the system widely open to multilingualism today, is to make it ready to face the future.

## 11. Concluding remarks

This paper is an attempt to define and justify the design of a digital system for traceability on a very long term. We have felt necessary to present an analysis in very abstract terms, axiomatic in spirit, and to connect each aspect with illustrations from a large range of cultural fields. It appears that embracing centuries consistently leads us to some fundamentals of cognition and cultural heritage, and perhaps paradoxically to the minute details of day-to-day work.

Early on, this diagnosis led one of us to seek a solution in operating system software research, and Plan 9 emerged as a prospect. After all, an operating system is a set of programs that make it easy for computer users and programmers to do their job [14]. We hope we have reached a sound, convincing outline, and identified some important essentials of working with a computer (and other sorts of memory devices!). Others should appear in the process of experimenting with a first implementation of the system. We hope that the design is clear, minimal and well-layered enough that we can cope with such surprises.

Our firm belief is that there is a favourable, exclusive niche position for Plan 9 on this topic, from where could emerge opportunities to spread like a virus. We submit that reflection to the Plan 9 community.

## 12. Acknowledgement

We thank Danny Lo Seen and Adrian Custer for reviewing early stages of this paper, Tristan Allouis for his commentaries on the figure. We are extremely grateful to Charles Forsyth who volunteered some editing, and ended up with a masterly rewrite. We found our words, translocated just in the right place. We are unfair not to count him as an author.

## References

[1] Mark Brown. *Archivists step in as Google Video shuts down for good.* http://www.wired.co.uk/news/archive/2011-04/18/google-video-termination, 18 April 2011.

[2] J. B. Buckheit, D. L. Donoho. *WaveLab and Reproducible Research.* Dept. of Statistics,

Stanford University, Tech. Rep. 474, 1995.

[3] DOI Factsheet. *Managing Data Relationships Using DOI® Resolution, Version 1.0.* http://www.doi.org/factsheets/ManagingDataRelationships.html, The International DOI foundation, accessed October 13, 2010.

[4] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, Julian Satran. *Object Storage: The Future Building Block for Storage Systems, a position paper.* In: proceedings of the Second International IEEE Symposium on Emergence of Globally Distributed Data, June 19-24, 2005, Sardinia, Italy.

[5] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, James W. O'Toole, Jr. *Semantic file systems.* In: Proceedings of the 3th ACM symposium on Operating systems principles, October, 1991.

[6] Donald E. Knuth. *Literate programming.* CSLI Lecture notes, 27, Center for the study of language and information, Stanford, California, USA, 1992.

[7] Kyong-Ho Lee, Oliver Slattery, Richang Lu, Xiao Tang, Victor McCrary. *The state of the art and practice in digital preservation.* Journal of research of the National institute of standards and technology, 107, 1, 2002, pp. 93--106.

[8] Bil Lewis. *Debugging backwards in time.* In: Proceedings of the Fifth international workshop on automated and algorithmic debugging (AADEBUG), Ghent, Belgium, September, 2003, Michiel Ronssea, Koen De Bosschere (Eds.), pp. 225-235. http://arxiv.org/pdf/cs.SE/0310016, [2010-10-14 Thu. 10:28].

[9] Robert C. Miller, Brad A. Myers. *Lightweight Structured Text Processing.* In: Proceedings of the 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA, pp. 131-144.

[10] Sean Quinlan, Sean Dorward. *Venti: A new approach to archival data storage.* In FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies, page 7, Berkeley, CA, USA, 2002. USENIX Association.

[11] Sean Quinlan, Jim McKie, Russ Cox. *Fossil, an archival file server.* Plan 9 user's manual, volume 2, fourth edition, 2002.

[12] Sean Rhea, Russ Cox, Alex Pesterev. *Fast, inexpensive content-addressed storage in Foundation.* In: Proceedings of the USENIX Annual Technical Conference 2008, June 22-27, Boston, Massachusetts, USA, pp. 143-156

[13] David S.H. Rosenthal. *LOCKSS: Lots Of Copies Keep Stuff Safe.* Presented to the NIST Digital Preservation Interoperability Framework Workshop, March 29-31, 2010.

[14] Jerome H. Saltzer, M. Frans Kaashoek. *Principles of Computer System Design: An Introduction.* Morgan Kaufman, Burlington, MA, USA, 2009.

[15] George Santayana. *Reason in common sense.* Dover Publications, Inc. New York, 1980 (first published by Charles Scribner's Sons, 1905).

[16] Denise Schmandt-Besserat. *Before writing, volume 1, from counting to cuneiform.* University of Texas Press, Austin, Texas, USA, 1992.

[17] Ben-Michael Schueler. *Update reconsidered.* In: Architecture and models in data base management: proceedings of the IFIP conference on modelling in data base management systems (Nice, France, 3--7 January 1977). G.~M. Nijssen, (Ed.), North Holland, Amsterdam, 1977, pp. 149-164.

[18] T. R. Schellenberg. *The management of archives.* Columbia University Press, 1965.

# Gostor: Storage beyond POSIX

*Latchesar Ionkov*
*Los Alamos National Laboratory*[*]
lionkov@lanl.gov

*ABSTRACT*

Gostor is an experimental platform for testing new file storage ideas for post POSIX usage. Gostor provides greater flexibility for manipulating the data within the file, including inserting and deleting data anywhere in the file, creating and removing holes in the data, etc. Each modification of the data creates a new file. Gostor isn't constrained by organizing the files in hierarchical structures, or identifying them with strings. Thus Gostor can be used to implement standard file systems as well as experimenting with new ways of storing and accessing users' data.

## 1. Introduction

Currently there are two popular ways of storing data – hierarchical file systems, and relational databases. The hierarchical file systems consist of **files** (arrays of bytes) that have names. The files are grouped in **directories** which can be members of other directories. In order to access a file, one needs to know the **path**, i.e. the list of directories from **root** of the file system. Files data can be overwritten, but new data can be added only at the end of the file. Most of the popular file systems support the POSIX (or similar) set of file operations [2]. Relational databases enforce strong structure on the stored data, by enforcing it to be fitted in a set of tables. The data can be accessed by using the SQL language. Relational databases provide better consistency and concurrency guarantees. They are hard to install and maintain, often requiring professional database administrator in order to achieve reasonable performance.

There have been many attempts to introduce the benefits of databases to file systems, such as transactions, snapshots, etc. Even though some popular file systems support snapshots and versioning, creating and manipulating snapshots is not part of the standard file operations.

In many cases, especially with scientific datasets, extending files only at the end is too restrictive. To avoid the limits, some scientific data formats, like HDF [1] re-implement most of the file system data structures in the file.

There are some attempts to go beyond both hierarchical naming of files as well as relaxation on the restrictions on how the content of the files can be modified. hFAD [4] allows data to be inserted and deleted at any place in the file. It also removes the dependency on file names when accessing the files and allows adding tags that describe the content of the files.

Most of the attempts to change the way data is stored and accessed are built on top of standard file systems or databases.

Gostor is a low-level storage system that provides consistent way of manipulating arrays of bytes, providing both versioning support as well as ability to insert, delete and modify data anywhere within the array. It also allows insertion and removal of holes in the array.

Gostor avoids the issues with hierarchical file names by not supporting them. All files in Gostor are immutable and can be accessed by a 64-bit ID that is returned when they are created. Files are created by cloning already existing files and modifying some of file's data.

[*]LANL publication: LA-UR-11-11422

## 2. Gostor Architecture

### 2.1. Files

A **file** in Gostor is an immutable string of bytes. Regions of file where data haven't been written are holes in the file. They are not stored and don't use disk space. Reading from a hole region returns zeroes.

A file is identified with a 64-bit integer called **fileID**. The fileID may change during the life span of the file. If the fileID is changed, Gostor guarantees that all references to the old fileID within the storage will be modified, and that the old fileID can be used until they are specifically discarded, or the connection to Gostor is closed. There is no guarantee that a fileID retrieved during previous connection will still be valid when a new connection is established.

File with fileID 0 always points to a zero-length file, and cloning it is how new files are created.

### 2.2. Data Content Types

When reorganizing the disk space, Gostor needs to know if any of the files contain references to other files. For that reason all data stored is assigned **data content type**. When writing data to files, the user can specify whether the data contains references to other files. Because data type is specified by write, there is no restriction for the whole file to have the same data content type. Gostor can coalesce content of data written by multiple writes as long as it is sequential and of the same type. When data is read, Gostor returns only data of the same content type.

Currently there are only two data types available to the user:

| Data Type | Description |
|---|---|
| **BTData** | Data that doesn't contain any fileIDs |
| **BTDir** | File content with only fileIDs |

Gostor defines additional data content types that are used internally to implement the file layout.

Gostor allows up to 32 thousand data types defined, but currently doesn't provide operations in the protocol for creating new data types. Extensions to Gostor (such as file systems) can use an internal API.

### 2.3. Operations

#### 2.3.1. Read

```
read(file, offset uint64, buf []byte) (count int, dtype uint16,
    err os.Error)
```

The **read** function reads up to length of the `buf` array bytes from the specified file, starting from the specified `offset` and stores it in the `buf` array. It returns the number of bytes returned as well as the data content type of the data. If an error occurs while reading, `err` contains the error.

Read can return less than `len(buf)` bytes in two cases – if there is no more data in the file (i.e. `offset + len(buf)` is greater than the file's size), or if the data in the file has mixed data content types.

#### 2.3.2. Write

```
write(file, soffset, eoffset uint64, count uint32, data []byte,
    dtype uint16) (newfile uint64, err os.Error)
```

Because Gostor files are immutable, each write operation creates a new file. It receives an existing file as argument, clones its content, applies the changes of the data content and returns the fileID of the newly created file.

The write call is used to do any modifications to a file. It replaces the data currently located between offsets (`soffset, eoffset`) with `count` bytes containing the data from the `data` array. If the length of the `data` array is less than `count`, a hole is created at the end of the region. Table 1 shows examples on how the **write** operation can be used to modify the data.

| Operation | Description |
|---|---|
| `write(.., 10, 10, 100, data,..)` | Insert 100 bytes of data at offset 10 |
| `write(.., 10, 10, 100, nil,..)` | Insert 100 byte hole at offset 10 |
| `write(.., 10, 110, 0, nil,..)` | Delete 100 bytes of data starting at offset 10 |
| `write(.., 10, 110, 100, data,..)` | Overwrite 100 bytes of data starting at offset 10 |

Table 1: Examples on data modifications using **write**

#### 2.3.3. Size

```
size(file uint64) (size uint64, err os.Error)
```

Returns the size of the specified file.

#### 2.3.4. Forget

```
forget(file uint64)
```

Informs Gostor that the user is no longer going to use the file with the specified fileID. If there are no more references to the file, it may be garbage-collected.

## 3. Current Implementation

Gostor is implemented in Go. Currently it is approximately 2000 lines of code.

### 3.1. Segments and Blocks

Gostor uses log-structured data layout, similar to the ones used in log-structures file systems [3]. The disk space is divided into segments, and the segments are kept in a doubly-linked list. The segments can be **free**, **full**, or **active**. There is only one active segment at a time. All new data is appended at the end of the active segment. Once the segment is full, it is marked as "full" and the next segment in the list is set as active.

The space within the segments is not divided into fixed-size chunks. Each segment consists of a number of variable-sized blocks. Each block can be up to $2^{16}$ bytes long (including the header). The header of the block contains its size as well as the type of the data it contains (data content type). The blocks always start at an even offset.

Figure 1 shows the physical layout of the segments and the blocks.

### 3.2. Files

Gostor uses modification of Btrees to describe files. Blocks at level 0 contain the data of the file. Blocks at level 1 contain entries to the data blocks at level 0, and so forth. Because Gostor allows insertion of data anywhere in a file, offsets to data are not constant and can't be stored in the intermediate Btree blocks. Instead, each Btree entry contains the size of the data it describes. Normally the Btree blocks are fixed size and waste some space when there are not enough entries. Because of the log-structured layout used by Gostor, the Btree representation we use is compact and doesn't waste any disk space.

The fileID of a file is the offset (relative to the beginning of the disk) of the block describing the root of the file Btree. This approach simplifies the implementation of Gostor and avoids a level of indirection, that is common to inode based systems.

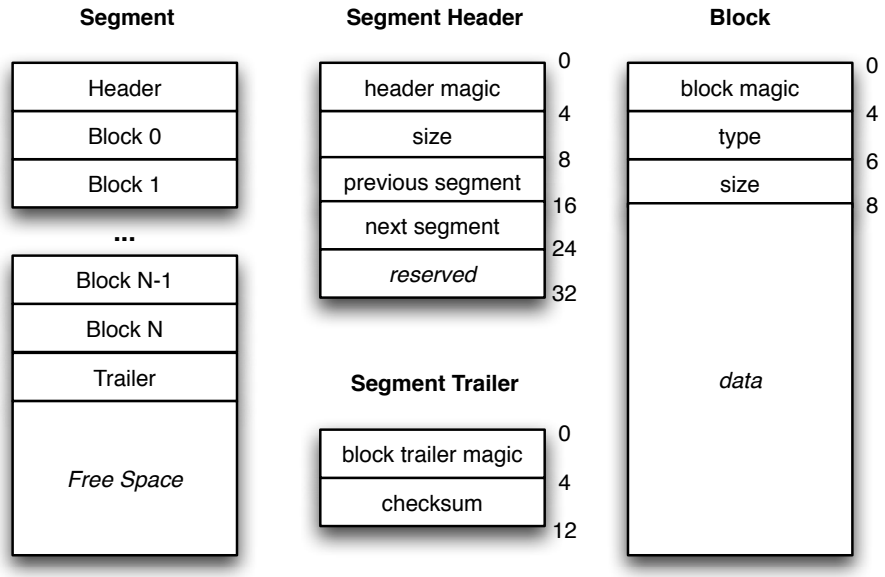



Figure 1: Gostor Disk Layout

Figure 2 shows an example of a file with fileID 348. The file's Btree has tree levels. The first entry in the root block points to block 388 and defines that that block describes the first 1100 bytes of the file. Block 388 has two entries, one is 700 bytes long and the data is stored in block 500. The second entry describes 400-byte long hole (the block is 0). Figure 2 also shows how the file blocks are laid out in a segment.

### 3.3. Operations

The **write** operation uses the specified file's Btree as a base for the Btree of the newly created file. It creates new blocks for a sub-tree of the file, up to the new file's root block. While building a new tree, Gostor tries to coalesce neighboring blocks if their size is less than the allowed size ($2^{16}$ bytes for level 0 blocks and $8192$ bytes for intermediate blocks). Gostor fills up the intermediate blocks with entries left-to-right so the rightmost affected block on the level is left unfilled. This approach is optimized for the most-frequently use case when data is appended at the end of the file.

Figure 3 shows the layout of the data when operation `write(340, 700, 700, 162, data)` is performed on the file shown on Figure 2. The operation inserts 150 bytes of data at offset 700 and a 12 byte hole. Gostor coalesces the hole with the already existing one, so no new entry for the hole is created.

### 3.4. Garbage Collection

Each **write** operation in Gostor creates a new file with a new root block, some intermediate Btree blocks as well as some data blocks. In most cases, once the new file is created, the old file is no longer useful and the blocks that belong only to it should be recycled. Gostor has knowledge of all references to blocks stored in it. It also keeps track of the blocks that the user is using at the moment. **Forget** operation should be called if the user no longer cares about a block.

Having information which blocks are still in use allows Gostor to reclaim the space that is no

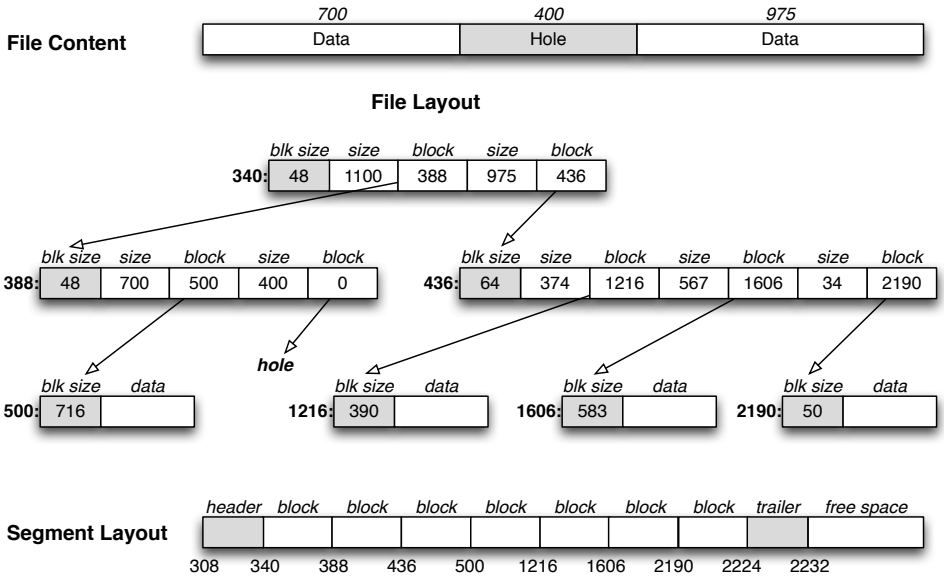



Figure 2: Btree and Segment Representation of a File

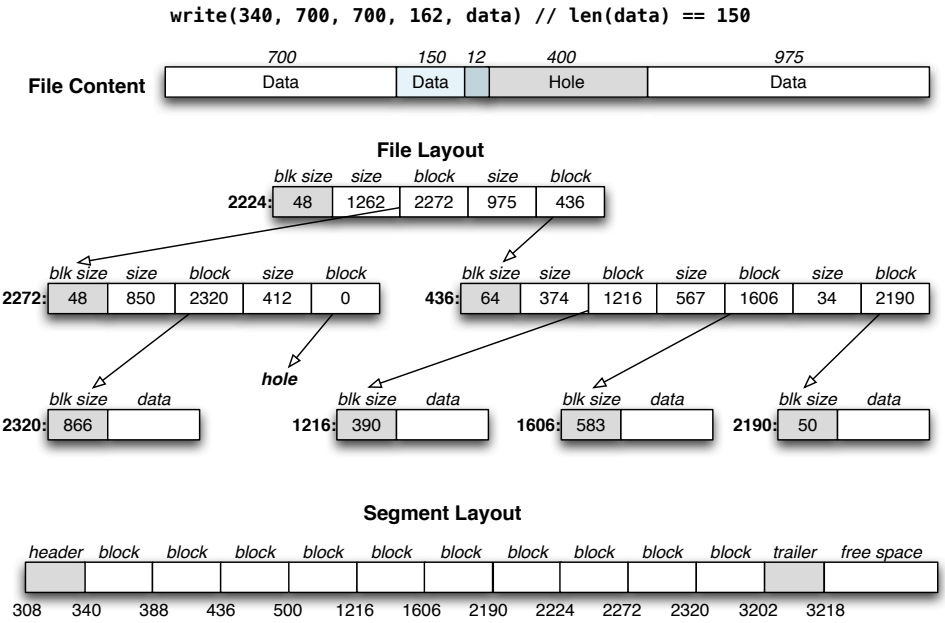



Figure 3: Btree and Segment Representation of a file after **write** operation

longer accessible. Currently Gostor supports only in-memory garbage collection. When a block is stored in Gostor, it is added to an in-memory list of data to be written to the disk. The data is kept in the memory for some time, giving it a chance to get obsoleted by subsequent changes to the list of live files.

When the user is no longer going to use a reference to a block, the **forget** command should be used to update Gostor's knowledge of what data is regarded as live by the user.

Currently Gostor supports only in-memory garbage collection. Once the data is committed to the disk, it can't be removed. Writing a garbage collector for the on-disk data is planned as future work.

### 3.5. Committing Blocks to Disk

When a block is stored in Gostor, it is not written to the disk right away. In most cases, the data in the block will be obsoleted soon by subsequent call and premature write to the disk would be wasteful. Gostor keeps the blocks in memory for some time to allow the data to be made obsolete. Gostore doesn't have knowledge what blocks will be actually stored on disk and therefore can't assign disk offsets for the blocks before they are actually committed. Instead, Gostor returns temporary offsets that can be used to access the data, or store references to the block in Gostor.

Before committing the data on disk, Gostor runs the garbage collector to free all blocks that are no longer live. Then it assigns disk offsets to the remaining blocks, and updates the temporary offsets stored in the blocks to the permanent ones. It can do that because it has knowledge of whether a block contains references to other blocks. Once the references are updated, the blocks are stored on the disk. The temporary offsets can be used until they are "forgotten".

## 4. Future Work

The current Gostor implementation doesn't have garbage collector that can reclaim data already written to the disk. There are plans for implementing copying, generational garbage collector that crawls through the existing data and copies the reachable blocks into new segments. Because of the high penalty of reading all existing data, the future implementation might be modified so it segregates blocks that contain references to other blocks in separate segments.

Once the Gostor prototype is stable enough, we plan to implement conventional hierarchical file system on top of it, as well as experiment with porting existing formats (HDF5) for scientific data on top of it.

## References


[1] HDF Hierarchical Data Format. http://www.hdfgroup.org.

[2] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). 1990.

[3] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 25:1–15, September 1991.

[4] Margo Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

# Revisiting User-Level Networking

*Jan Sacha*
*Jeff Napper*
*Henning Schild*
*Sape Mullender*

Bell Laboratories
Antwerp, Belgium

**Abstract**

User-level networking has been applied mostly in the area of high-performance computing since it allows applications to obtain maximum performance from the hardware. We argue that running network protocol stacks in the user space also allows applications to reduce the amount of state that the operating system kernel maintains on their behalf enabling much greater application elasticity and mobility. We describe a lightweight user-level networking framework that does not require any modification to existing networking hardware nor protocols. The main advantages of our framework are its simplicity and ease of implementation.

## 1 Introduction

User-level implementations of network protocol stacks have existed for many years. Initially, such stacks were used to develop and test novel network protocols which ran outside of the operating system kernel and thus did not harm the kernel's reliability. They were also easier to deploy since they did not require kernel recompilation and reboot and were easier to debug [12, 17]. With the advent of high-performance computing, user-level networking also became a mechanism to increase application performance. By bypassing the kernel and interacting directly with the network interface controller, applications could reduce their overhead and achieve lower communication latency [18, 13, 4].

We claim that there exists yet one more reason to implement network protocol stacks in the user space. As applications gradually move to the cloud, it becomes increasingly important to be able to checkpoint, suspend, resume, marshal, and migrate applications. All these tasks are greatly simplified if the operating system kernel maintains very little or no state on behalf of user processes. This state, however, is very often stored by kernel-level network protocol implementations, for example in the form of connection parameters and acknowledged data buffers. In order to support application elasticity and mobility, the operating system needs to be able to extract application state from its internal structures and consistently redeploy it when needed. By running the entire network stack in the user space, not only do applications limit their kernel-level state but also gain an opportunity to react to mobility events in an application appropriate way.

Further, we argue that running network protocol stacks inside user applications naturally improves modern hardware utilization. According to the current trends in computer architecture evolution, both the core count per host and network bandwidths grow at a high rate. In order to utilize hardware efficiently, network traffic needs to be processed in parallel on multiple cores, which can be achieved by dispatching packets from the network interface to the application cores as early as possible. Such a design also improves memory cache performance because each packet is processed mostly by one core only. Finally, high-performance applications might tune the behavior of their network stacks using application-specific knowledge in order to achieve better performance.

Many existing approaches to user-level networking require either using specialized networking hardware or running customized network protocols. In this paper we describe a light-weight framework for user-level networking that runs on generic hardware and does not require any modifications

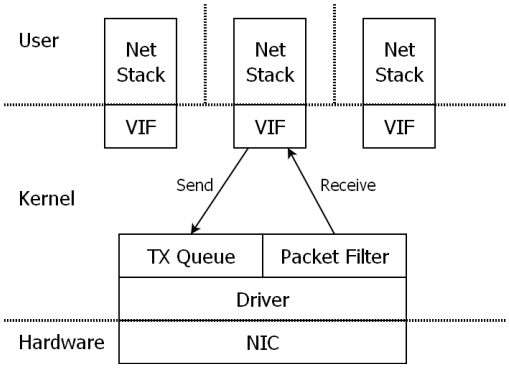



Figure 1: System overview

to the existing network protocols. The main advantages of our framework are its simplicity and low amortized overhead. Our approach is based on the notion of virtual network interfaces that isolate access to physical interfaces from user applications. A virtual interface is defined using two bitmask patters that describe incoming and outgoing packets. Using these patterns, an efficient demultiplexing algorithm dispatches packets received from the physical network interface directly into user applications. The network protocol stacks are implemented entirely in the user space.

The rest of this paper is organized as follows. We discuss related work in section 2, we describe our framework in section 3, and we evaluate it in section 4.

## 2 Related Work

The earliest approaches to user-level networking focused mostly on the flexibility and ease of implementation for new network protocols rather than performance. They usually provided built-in packet filtering languages [12, 11] or allowed user processes to register custom packet handlers in the kernel to filter the incoming traffic [19]. Some of these built-in languages were later optimized using dynamic code generation [20, 7].

In the high-performance computing area, user-level networking was used as a mechanism to eliminate the kernel from the critical path when sending or receiving network packets. The main challenge in these approaches, apart from maximizing performance, was to provide user isolation and security. Most proposed solutions either relied on custom communication protocols, such as Active Messages [19], Fast Messages [13], and BIP [14], or required packet tagging [18]. Other high-performance solutions relied on specialized hardware [17, 5]. These later approaches eventually lead to a standard, the Virtual Interface Architecture [6, 4], which defined a set of hardware functions needed to support secure and efficient user-level network interface access.

A large research effort was devoted into designing packet classification algorithms for applications such as routing, admission control, intrusion detection, and traffic accounting [8, 10, 15]. It was shown that the complexity in a general packet classification problem grows exponentially with the number of packet classification rules and thus the search space might become extremely large even for moderate rule sets. Most state-of-the-art algorithms address the classification problem by constructing decision trees that allow packet processing with a minimum number of memory accesses. Due to the search space explosion, such decision trees often require very large amounts of memory [15]. Our approach is closer to the PathFinder [2] and the Tuple Space Search [16] algorithms which use bitmask patterns and hash tables and typically require much less memory.

A separate approach to couple the network protocol stack with the application is to use a virtual machine such as Xen [3] or KVM [9]. Typically, the hypervisor provides to each guest system a virtualized network interface which either has its own unique MAC address and is bridged to the physical interface or has the same MAC and IP addresses as the physical interface and is multiplexed using Network Address Translation (NAT). The main disadvantage of these approaches compared to our model is that the hypervisor and the guest operating system add a significant overhead and complexity to the user application.

```
struct Packetdigest{
        struct{ /* Ether */
                uchar   src[6];
                uchar   dst[6];
                uchar   type[2];
        };
        union{
                struct{ /* IPv4 */
                        uchar   src[4];
                        uchar   dst[4];
                        uchar   proto[1];
                        /* TCP or UDP */
                        uchar   sport[2];
                        uchar   dport[2];
                };
                struct{ /* ICMP */
                        uchar   type[1];
                        uchar   code[1];
                };
                struct{ /* AoE */
                        uchar   major[2];
                        uchar   minor[1];
                        uchar   tag[4];
                };
        };
};

struct Packetpattern {
        struct Packetdigest     mask;
        struct Packetdigest     value;
};
```

Figure 2: Sample packet digest definition in the C language

# 3 System Model

An overview of our networking framework is shown in figure 1. Every physical network interface (NIC) is associated with a collection of *virtual interfaces* (VIF) which allow user processes to send and receive packets. Typically, network protocol implementations generate a virtual interface per each communication endpoint such as a TCP or UDP socket. A virtual interface consists of two bitmask-value patterns, which describe the packets that the interface is allowed to transmit and receive, and a kernel-level queue for incoming packets. The purpose of the bitmask-value patterns is twofold: to verify packets generated by processes and to demultiplex incoming traffic. We also envisage that virtual interfaces will enable the kernel to control the network resource consumption by users processes. For example, the kernel could divide available bandwidth between virtual interfaces according to some policy or provide low-latency network access to real-time processes.

In order to send a packet, a user-level network stack generates all packet headers (including data layer) and passes the packet payload together with the headers to the kernel. The kernel verifies that the packet matches the allowed pattern and attempts to transmit it. If the NIC is busy, the packet is appended to the driver's transmit queue. Normally, patterns for outgoing packets are not allowed to overlap so that processes cannot interfere with each others traffic. If a process attempts to register a virtual interface with a pattern overlapping with another virtual interface, which might for example happen if the process tries to open a connection on a port used by another process, the kernel returns an error.

When an incoming packet is received from hardware, the packet filter compares the packet headers with the bitmask-value patterns defined in the virtual interfaces and selects the virtual

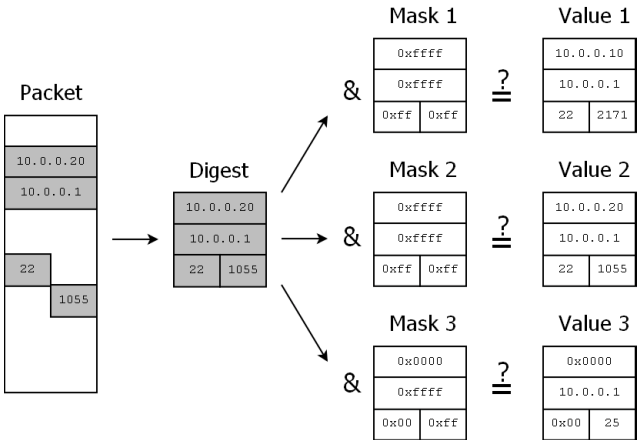



Figure 3: Sample packet digest and bitmask-value patterns

interface that receives the packet. If a user process is already waiting for a packet from the selected virtual interface, the kernel delivers the packet directly to the user-space network protocol stack. If no process is waiting for the packet, the kernel enqueues the packet in the virtual interface.

## 3.1 Packet filtering

Network access control and traffic demultiplexing is performed using a data structure called *packet digest* which contains all the fields extracted from packet headers that are relevant for packet filtering such as protocol types, source and destination addresses, and port numbers. Figure 2 shows a sample packet digest format in the C language. Since the digest structure is shared by all protocols it contains a union of alternative fields at each protocol layer. Only the data layer is fixed (Ethernet in the discussed example) since it is usually NIC-specific.

Each virtual interface defines two packet patterns: for incoming and outgoing packets. Each such pattern is a pair of packet digest structures, where the first structure is interpreted as a binary bitmask and the second is the corresponding binary value. Note that bitmask-value expressions are sufficiently flexible to represent common pattern types such as exact values for fields and subfields, wildcards (i.e., zeroes in the bitmask) and prefixes (particularly useful for IP subnets). Arbitrary ranges (e.g., port ranges) can be represented by converting them to sets of prefixes.

In order to verify if a packet matches a bitmask-value pattern, a digest is first generated from the packet by extracting relevant fields from the packet header. Since lower-level protocol headers always define the type of the higher-level protocol header such a digest is always unambiguous. A packet digest $D$ matches a bitmask-value pair $(B, V)$ of some virtual interface if $D \,\&\, B = V$ where $\&$ is a bitwise AND operator. Again, note that although the packet digest structure contains a union of overlapping fields, packets sent by different protocols (and hence using different fields in the header) always differ in the type field of the lower-layer protocol and thus the bitmask matching algorithm always classifies them correctly.

In order to dispatch a packet received from hardware to a virtual interface, the packet filter generates a packet digest and iteratively tries to match it against receive patterns of all virtual interfaces associated with the NIC. The first virtual interface that matches the digest receives the packet. If no pattern matches the digest, the packet is dropped (see figure 3).

In a simple system configuration virtual interfaces belonging to the same NIC should not have overlapping patterns. If for some reason processes need to create interfaces with overlapping receive patterns, for example to express some complex filtering rules, the packet filter must know the order in which these patterns are applied. If an incoming packet matches multiple virtual interfaces, the first matching interface receives the packet. Optionally, the packet filter may copy the packet and deliver it to multiple matching interfaces, which allows implementing tools such as network sniffers.

Note that an alternative packet filter design is possible where packet digests are not generated but instead mask-value pairs are directly applied to the packet headers. However, packet headers

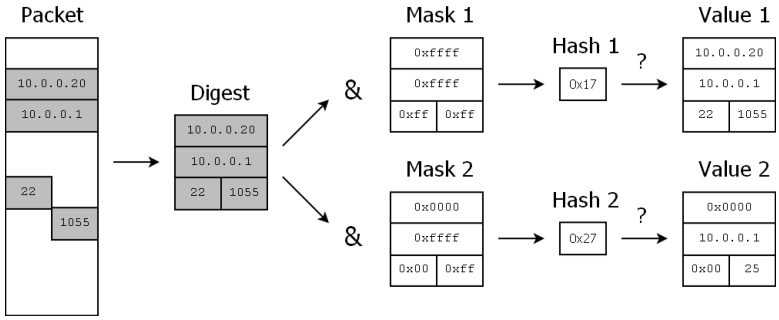



Figure 4: Packet filtering using hash tables

normally contain many fields that are not relevant for packet filtering and thus their comparison would require unnecessary memory accesses. In a typical packet digest definition, such as the one shown in figure 2, a packet digest occupies less than 32 bytes of memory and thus a bitmask-value comparison can be performed by fetching one cache line on most modern architectures. Further, some protocols have variable-length headers (e.g., options) that cannot be classified using simple bitmask-value expressions. On the other hand, generating packet digests requires the kernel to be able to parse protocol headers. We assume though that the kernel needs to have some protocol-specific knowledge for fairness and security reasons.

## 3.2 Optimizations

The packet demultiplexing algorithm described above requires matching an incoming packet digest with the patterns of potentially all virtual interfaces associated with the NIC. Hence, the algorithm has a linear complexity with the number of virtual interfaces.

One way to improve packet demultiplexing efficiency is to sort the virtual interfaces on the NIC's list based on the number of packets they receive. The packet filter could for example increment a counter in a receiving virtual interface each time it forwards a packet. Occasionally, the packet filter would also re-sort the virtual interface list based on these counters. When processing an incoming packet, the packet filter would then try to match against interfaces that are most likely to receive the packet, increasing the probability of a hit. Given that traffic distributions are often very skewed, such an optimization could yield in a constant amortized filtering time for packets that match interface patterns. However, for packets that do not match any virtual interface (and are thus discarded), the packet filter would still have to iterate over all interface patterns before making a decision and would thus require linear processing time.

Another optimization is based on the observation that many virtual interfaces are likely to have exactly the same bitmasks. For example TCP/IP connections are distinguished by the source and destination IP addresses and source and destination port numbers. Thus, all virtual interfaces corresponding to TCP/IP connections generate the same bitmasks but differ in the associated values (i.e., different port numbers or IP addresses). In order to forward packets efficiently, the packet filter could group all virtual interfaces that share the same bitmask and store their values in a hash table. When processing a packet, the packet filter would calculate the digest, apply a bitmask, calculate a hash from the masked digest, and perform a lookup to check if any value associated with the bitmask matches the digest (see figure 4). Given a sufficient number of bins in the hash table, this filtering method would allow packets to be processed in a constant time per protocol bitmask.

However, pattern matching using hash tables has one limitation. If patterns are allowed to overlap, the order in which patterns are applied to packets may be relevant for the filtering semantics. Grouping patters together using hash tables may thus affect this order. For example, consider a sequence of three patterns that need to be matched in the following order: A, B, and C. Suppose that patterns A and C have the same bitmask and are inserted into a hash table. In such a case, it is impossible to preserve the original pattern matching order because either an A-C hash table lookup is performed before applying pattern B, in which case packets matching both

| Trace | Rules | Before hashing | | After hashing | | Packets |
|---|---|---|---|---|---|---|
| | | Bitmasks | Values | Bitmasks | Values | |
| acl1 | 751 | 79 | 1,246 | 81 | 1,250 | 8,140 |
| acl1_100 | 98 | 28 | 129 | 28 | 129 | 1,000 |
| acl1_10K | 9,603 | 109 | 12,931 | 109 | 13,044 | 97,000 |
| acl1_1K | 916 | 68 | 1,222 | 68 | 1,222 | 9,380 |
| acl1_5K | 4,415 | 98 | 6,109 | 99 | 6,139 | 45,600 |
| fw1 | 269 | 221 | 914 | 644 | 3,623 | 2,830 |
| fw1_100 | 92 | 173 | 302 | 185 | 381 | 920 |
| fw1_10K | 9,311 | 238 | 32,136 | 382 | 998,366 | 93,250 |
| fw1_1K | 791 | 224 | 3306 | 263 | 3,420 | 8,050 |
| fw1_5K | 4,653 | 235 | 15,778 | 359 | 22,751 | 46,700 |
| ipc1 | 1,550 | 244 | 2,180 | 624 | 8,329 | 17,020 |
| ipc1_100 | 99 | 54 | 145 | 55 | 146 | 990 |
| ipc1_10K | 9,037 | 310 | 12,127 | 388 | 16,027 | 90,640 |
| ipc1_1K | 938 | 186 | 1,223 | 190 | 1,251 | 9,380 |
| ipc1_5K | 4,460 | 276 | 5,916 | 311 | 8,430 | 44,790 |

Table 1: Traces used in the experiments

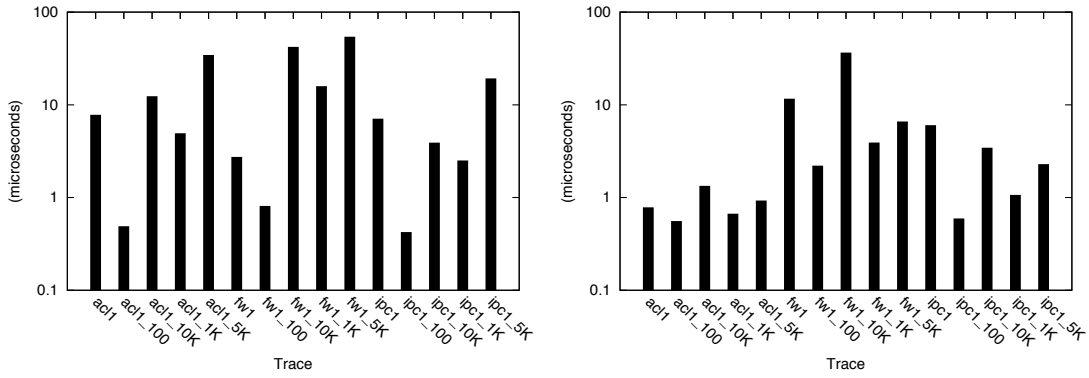
B and C may be classified incorrectly, or pattern B is matched before the hash table lookup, in which case packets matching both A and B may be classified incorrectly.

A solution to this problem is to explicitly add filtering rules for packets that match multiple patterns. In the example above, one could add a new pattern, applied first, for packets that match both A and B. The packet filter would then apply pattern B and at the end it would perform a lookup in the A-C hash table. This solution increases the total number of patterns but it preserves the original filtering semantics.

### 3.3 User-Kernel Transition

In order to achieve high throughput, the networking code should avoid unnecessary packet copying. While the implementation of the protocol stacks in our framework is entirely application dependent, it is an interesting question whether the kernel can avoid packet copying when sending and receiving packets from the user space. In principle, the kernel can translate the virtual address of the user-space outgoing packet to (possibly multiple) physical addresses and pass them to the NIC for transmission. For security reasons, it might copy the packet header so that the user is not able to modify it. However, some networking cards have limitations on the memory buffers they can use. For example, some NICs can only access a subset of host's memory due to addressing limitations, or have restrictions on the buffer alignment, or do not have a scatter-gather capability and can only transmit packets that are contiguous in the physical memory. If the networking card has one of such limitations, the user either has to allocate packets in a special way (e.g., by asking the kernel) or the kernel has to copy the packet from the user space to a kernel buffer before transmission.

Delivering an incoming packet from the kernel space to the user space without copying is even more problematic. Theoretically it can be done by changing the virtual memory mapping in the receiving process. However, virtual memory can only be modified at a page granularity. If the receiving process requests the packet to be delivered to a specific buffer, both the buffer and the packet payload must be aligned on a page boundary. In order to assure such a packet payload alignment, the kernel would need to know in advance the header length of the packet it is going to receive from the network. We are currently investigating whether it is possible to control incoming packet alignment by assigning MAC addresses or VLAN identifiers to data-intensive network protocols.




(a) Linear pattern matching

(b) Hash table optimization

Figure 5: Average packet demultiplexing time

# 4 Evaluation

In order to get a preliminary feedback on our framework's feasibility, we implemented the bitmask-based packet filtering algorithm on Plan9 and fed it with traces of packets and filtering rules obtained from a public trace archive [1]. The traces contain three classes of rule sets: Access Control Lists (ACL), Firewall rules (FW) and IP Chains (IPC) summarized in table 1. Columns 2 and 3 show the number of unique bitmasks and values produced from each rule set. Note that due to the range conversion the number of bitmask-value patterns is usually higher than the number of rules in the trace. Furthermore, the number of patterns is increased by the hashing algorithm (columns 4 and 5) which creates new filtering rules for patter intersections in order to preserve the original rule matching order. For some rule sets the number of extra patterns is notably high.

We ran our packet filtering algorithms on an AMD K10 Opteron machine on a single 2.2 GHz core. We first read all the packets from the trace to the main memory and then measured the total time needed to classify all packets. Figure 5 shows the average packet filtering time using a linear pattern matching algorithm and a hash table based algorithm. Clearly, in most cases the hash table optimization improves performance. For most traces, the average packet filtering time is on the order of microseconds. Given a packet size of a few kilobytes, this would allow filtering traffic of approximately one gigabyte per second per core or equivalently 10 gigabits per second per core. Since our prototype is not aggressively optimized (e.g., we do not use SIMD instructions) the throughput may potentially be improved by a more efficient implementation.

# 5 Conclusions

In this paper we argue that for future applications it will be advantageous to run network protocol stacks in the user space. Many-core machines will need to be able to dispatch incoming packets as quickly as possible to the receiving application cores to maximize performance. We describe a simple user-level networking framework which requires only very limited protocol-specific knowledge in the kernel and allows running entire network protocol stacks in the user space. Measurements on an experimental prototype show that our approach is viable.

# References

[1] `http://www.arl.wustl.edu/ hs1/PClassEval.html`.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *OSDI*, pages 115–123, 1994.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177. ACM, 2003.

[4] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. In *ACM/IEEE Supercomputing*, pages 1–15, 1998.

[5] C. Dubnicki, L. Iftode, E. W. Felten, and K. Li. Software support for virtual memory-mapped communication. In *International Parallel Processing Symposium*, pages 372–281. IEEE, 1996.

[6] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18:66–76, 1998.

[7] D. R. Engler and M. F. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59. ACM, 1996.

[8] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15:24–32, 2001.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Linux Symposium*, 2007.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *SIGCOMM*, pages 203–214. ACM, 1998.

[11] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *USENIX Winter Conference*, pages 2–12, 1993.

[12] J. Mogul, R. Rashid, and M. Accetta. The packer filter: an efficient mechanism for user-level network code. In *SOSP*, pages 39–51. ACM, 1987.

[13] S. Pakin, V. Karamcheti, and A. A. Chien. Fast messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Parallel & Distributed Technology*, 5:60–73, 1997.

[14] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on myrinet. In *IPPS SPDP Workshops*, pages 472–485, 1998.

[15] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li. Packet classification algorithms: From theory to practice. In *INFOCOM*, pages 648–656, 2009.

[16] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *SIGCOMM*, pages 135–146. ACM, 1999.

[17] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1:554–565, 1993.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP*, pages 40–53. ACM, 1995.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, pages 256–266. ACM, 1992.

[20] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter Technical Conference*, 1994.

# Acid your ARM

*Gorka Guardiola Múzquiz*
*Laboratorio de Sistemas*
*Universidad Rey Juan Carlos*
paurea@lsub.org
*9/1/2011*

*ABSTRACT*

We have developed `jtagfs`, a protocol stack and filesystem which enables live debugging of an ARM machine using acid. It accesses the hardware through a JTAG interface, providing new ways of debugging which dissolve the boundaries between the kernel, user space and the loader and gives direct access to the hardware. At the same time acid provides high level abstractions to interpret the results and automate the debugging process.

## Introduction

JTAG is a standard for boundary scanning, a method for testing digital circuits by means of a shift register (boundary scan shift register or BSR). The BSR is used to drive the inputs and outputs of different parts of the circuit. On each subcircuit the BSR is controlled by a TAP (Test Access Port), used to transverse the states of the BSR on each tick of the clock, load it and set its connections to the pins, input and output values of the circuit. On complex digital circuits, like a microprocessor, the BSR can be used to control separately and test different subcircuits by feeding it different chains (a sequence of bits).

While debugging some drivers in the Sheevaplug, it came to our attention that the ARM cores have a well documented JTAG interface [10]. Furthermore, the chains to control all of the models are very similar, with only small differences between them. The microprocessor includes a TAP controller and some extra debugging hardware as part of the macrocell called Embedded ICE [6] or Embedded ICE-RT [7] depending on the particular model. The JTAG interface provides access from an external external machine to different parts of the microprocessor. Through JTAG the debugger can make the processor enter debug mode, write directly to the processor registers, inject instructions with full access to the hardware and restart the processor no matter its state. Furthermore, some machines like the Sheevaplug contain a chip which in addition to providing access to the serial port on the machine has a subcircuit able to interact with the TAP controllers on the board and the microprocessor, making them accessible through USB. This chip, called AN2232C-01 [4] , is a command processor which can drive any kind of serial interface (act as an MPSSE or Multi-Protocol Synchronous Serial Engine). It can also work as an MCU host bus emulator. We will be using it as an MPSSE, so we will just call it MPSSE from now on.

All this capabilities mean that with the appropriate software it is possible to debug the kernel having facilities akin to that available on special development boards while at the same time running regular production kernels. It is even possible to debug simultaneously the kernel, user space and the loader, vanishing the frontiers and providing full access to the hardware. The problem is that the appropriate software did not exist. The software we have had access to has some limited debugging capabilities through gdb or direct access to the hardware. Porting existing software to Plan 9, while being more complicated than writing it from scratch would have been not enough because of the dependencies with gdb and its lack of generality in the interfaces it offered. Leveraging on [14] and with an approach similar to `rdbfs`(4) [11] but with a twist, we have developed a general purpose a complete programmable debugging interface for the ARM machines, providing full access to the hardware.

### JTAG basics

Each JTAG capable device has a number of TAPs connected in series or in parallel. Each TAP normally has four inputs, TCLK, TMS, TDI and TDO, connected as is shown in Figure 1.

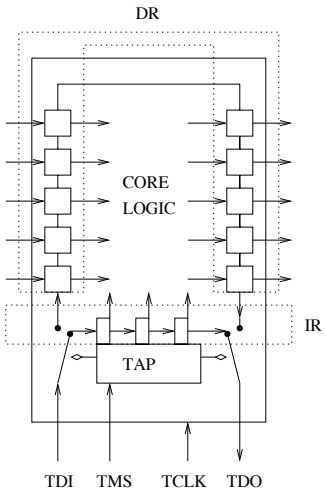



**Figure 1:** *Jtag enabled device*

On each TCLK down edge the system may shift the value in TDI and shift out the value to TDO depending on the state of the state machine of the TAP, depicted on Figure 2, with the transitions controlled by the value in the TMS input. There are two shift registers normally connected in parallel, the data register (DR) and the instruction register (IR). The DR is the BSR, and the IR controls what happens. Which of them is connected to TDI and TDO depends on the state of the TAP controller which also sets when the instructions or the data start being active and connected to the chips or the output pins. TAP controllers can be chained in series, with the data registers and the instruction registers concatenated. There is an instruction to disable a controller which can effectively turn off one controller so that data can be shifted in separately.

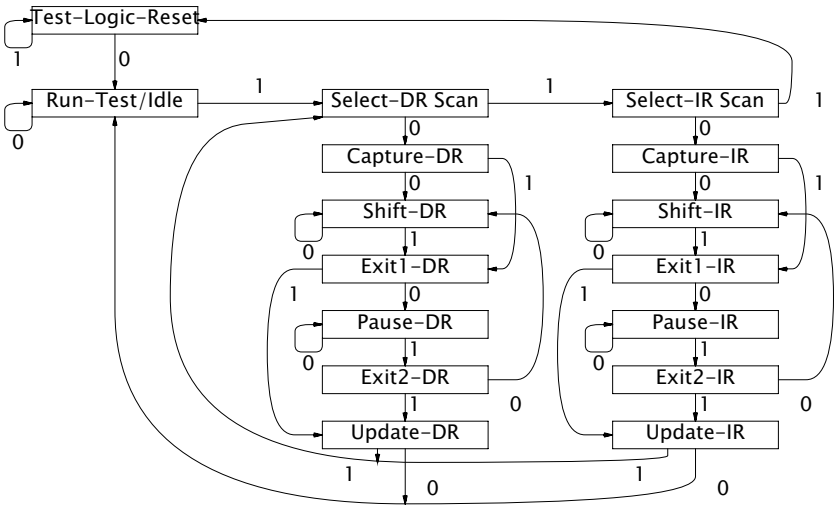



**Figure 2:** *JTAG TAP state machine, input is TMS*

Sometimes controllers are also chained in parallel where there are different chains and an instruction is used for chain selection. For example this approach is taken by the ARM to select between the different circuit modules. Each chain provides a length for the instruction register and the data register.

The instructions supported by the IR may vary among models with some of them being mandatory. For more details on this, see the JTAG [10], though the details of what is implemented and what instructions are supported are actually detailed in the ARM manuals.

The instructions supported by all ARM machines are (there are some minor differences in semantics):

•SCAN_N is used to input a chain number.

•INTEST is used to set the chain number input with SCAN_N.

•IDCODE is used to detect the chip and puts a special ID value in the DR.

•BYPASS disables the TAP, putting a 1 bit shift register between input and output.

•RESTART restarts the processor after it entered debug state.

### Architecture of jtagfs

Figure 3 depicts the architecture for the `jtagfs`.

The first part which needed to be implemented was the access to the MPSSE in the FTDI chip. There was already partial support for the FTDI chip in `usb/serial`(4) (for more details see [12] and [1]). We completed the support for the FTDI configuration protocol, but kept the MPSSE support itself outside of it. The `usb/serial` is already complicated enough. We set the usb serial chip with the minimum possible configuration (set the interface to be MPSSE and the latency timer) and made it serve a file called jtag which serves as conduit to communicate through it. The `jtagfs` uses this file to send commands to the MPSSE.

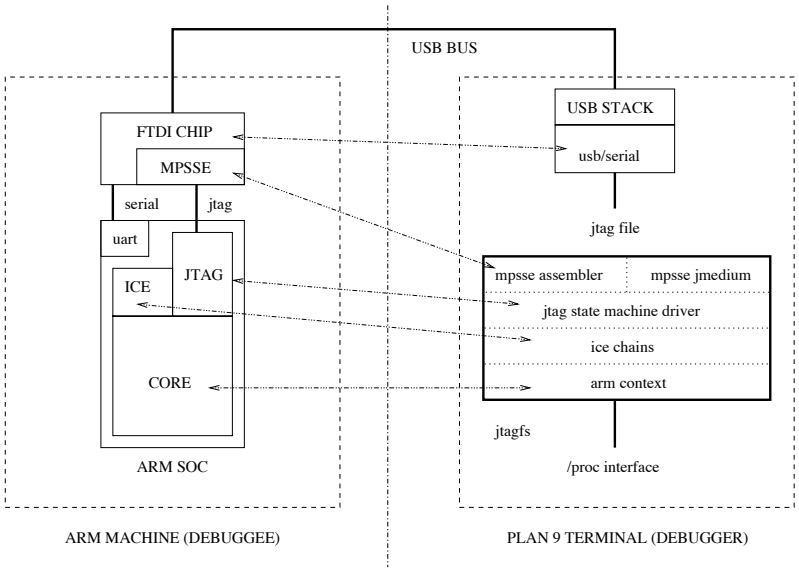



**Figure 3:** *Jtagfs architecture*

The next level of abstraction is the `JMedium`, a data type and a set of operations which abstracts the details of the driver for the JTAG. For now, we use an MPSSE implementation for the `JMedium`, but others may be implemented in the future. Using it, we just drive the TDI, TDO, TMS with respect to the TCLK. The interface uses buffering and takes in account there may be more than one TAP connected in series, though all but one (the current TAP being drived) are disabled.

We also wrote a state machine driver, which given an origin state and a destination state calculates the minimum distance path between them. Given the number of states, a static table could be precalculated to do this when compiling, but it was not done for simplicity, given that most of the time is spent waiting for the USB communications anyway.

The next level is the ICE chains layer, which takes care of the strange bit endianness of the JTAG chains on the ARM (some of the chains are bit order inverted and byte shuffled), and converts between that endianness and the local (debugger) machine byte order.

Built on top of the ICE chains, there is a level which interacts with the ARM processor, making it enter and exit debug mode, saving and restoring the registers and injecting instructions.

Finally, a `proc`(3) interface exports the processor registers in a manner similar to `rdbfs`(4). A small library for acid and a little modification to `attachproc` in the `mach`(2) library to make the kernel registers file, `kregs`, writable makes it possible to switch modes and manipulate the processor at will from acid.

### Driving the MPSSE

The MPSSE itself is quite a complex device which can drive any kind of serial interface. It is programmed through a small machine language which can output bits in different bit endianness, driving TMS, TDI and TDO on down or up TCLK edges. We started outputting the machine commands directly, but, specially while debugging it was quite complex to keep track of all the details of the MPSSE. We ended writing a small assembler for the MPSSE instructions, which we called `ma` and assembling them on the fly. `Ma` is a good name because we will never really have `.m` object files (for which the letter `m` is already taken), so there will not be another assembler with this name, at least in kencc. An example of the instructions can be seen next:

```
DataIn EdgeDown LSB 3
DataIn EdgeDown LSB B3
DataOutIn EdgeDown EdgeUp LSB 3 0x42 0x34 0x56
DataOutIn EdgeDown EdgeDown LSB 3 @
DataOutIn EdgeDown EdgeUp LSB B3 0x42
DataOutIn EdgeDown EdgeDown LSB B3 @
TmsCsOut EdgeDown MSB B0x7 0x7
TmsCsOut EdgeDown LSB B7 0x7
TmsCsOutIn EdgeDown EdgeUp LSB B0x7 0x7
MCURd 0x34
WaitIOHigh
AdaptClkDisab
Div5ClkEnab
Loop
SetBitsL 0x32 0x34
```

`MSB` and `LSB` mean most and least significant bit first, `BNN` means a number of bits (whereas the count by itself means a number of bytes) and `@` is used as a parameter when assembling on the fly as a placeholder for the data (passed as another parameter).

Assembling on the fly has proved to be a very good approach for debugging and testing, providing a low level sniffing interface. In the first prototype, we hardcoded the values using constants and some functions. Each time we found a bug in our interpretation of the MPSSE instructions, we had to fish bugs on every place where they happened. Also, the assembler itself may be useful for other applications using this chip for driving other (for example SPI) interfaces.

### ICE chain support

There are several chains on each ARM machine, which provide access to different capabilities of the chip. There are minor differences among them, and some of the chains are present on some chips and not on others. We have interacted mainly with three chains.

Chain 1, is used to inject instructions to the ARM core. Special care needs to be taken with the clock. Basically, when the processor is in debug mode (which is when an instruction can be fed to the core) it runs on a slower clock. As a consequence, whenever an instruction to interact with external hardware, like RAM or a peripheral, needs to be executed, the processor must run it using the faster clock. Then it falls back to the slow clock driving the debug mode.

Chain 2, is used to access the debug registers, most of which can be accessed normally from inside the core. This registers enable hardware vector catching (enter debug on an exception, including reset), breakpoints (entering debug mode based on an address being executed) and watchpoints (entering debug mode based on an address being read or written) and instantly entering and exiting debug mode.

While chain 1 and chain 2 are well documented and seem to be the same on all the latest ARM cores, Chain 15, which provides access to the MMU, seems vary more from model to model. From what we have seen, there are two families of ARM with respect of Chain 15, the ARM 7 family and the ARM 9 family. In any case, we could not made Chain 15 work on the feroceon, so instead we have pushed MCR and MRC instructions. This approach is more portable and without any drawbacks. Using them we added the MMU state to the observable state of the processor. This state is read only at the moment, though this can change in the future.

### ARM interaction

The ARM processor interaction code has two different levels. For example, there is a function, `ARMgofetch` used to inject an instruction into the core. This instruction is pushed into the pipeline and goes through the five states (fetch, decode, execute, access an writeback). At this level, one has to be careful what state the pipeline for the instruction is for. To abstract the pipeline we wrote some other functions (for example `ARMgetexec` and `ARMsetexec` ) which shift in the instruction, inject NOPs and read and write the data when the pipeline is in the right state. They also make sure that after the pipeline is full until the instruction finishes.

We found that the litmus test to find whether the whole system works is if the processor is able to run again after going into debug mode. All the context for the Arm needs to be perfect. In this respect, we found two difficulties while implementing `jtagfs`. The first one is that even if the PC does not need to change, the processor will not start if the register is not written to. The second difficulty, is that interrupts need to be disabled while in debug mode. If they are not disabled, bad things will happen. An interesting consequence of this is that if while in debug mode something improper is done, like access a non mapped address, when we start the processor again, an interrupt will fire that will most probably crash the system.

### Endianness

Endianness in the JTAG is tricky. There are two interfaces, the `proc`(3) on one side and the JTAG on the other side and both need to be honored. The `proc`(3) interface should be in ARM endianness (little endian on Plan 9), whereas the JTAG has special bit ordering, which is different for every chain, but at the same time the little endianness of the ARM needs to be respected. The approach we have taken is that the registers, which need to be modified by jtagfs are translated to host order at the interface ( `proc`(3) and chain interface). On the other hand, other data passes through without going into host order.

### Filesystem

The standard proc filesystem is used to export processes. The model was extended by `rdbfs`(4) and the `-k` flag for acid to provide access to the segments and registers of a running kernel. `Jtagfs` extends this model even further, exporting more registers (in particular exporting the MMU registers) and mapping memory outside of the segments of the binary.

The MMU registers can be accessed using the `regs` file, just after the `Ureg` and floating point registers (if there are any). To take advantage of them, the jtag acid library uses the undocumented `map`() builtin for acid to extend the register map. This approach makes it possible, when the processor is stopped, to access the memory

mapped MMU registers from within acid.

The other extension implemented is that the acid library also uses `map()` to extend the memory mapped for the data segment to all the memory starting from KZERO. This approach lets us access memory outside the kernel segments, like `Mach` and the page tables.

Thanks to these extensions, with very little code, it is possible translate from virtual addresses to physical addresses by looking them up in the current page table.

### Debugging

Debugging the `jtagfs` was a challenging activity. Running wireshark under linux to capture the USB dialog of OpenOCD proved invaluable in the first stages, specially to understand the finer points of the timing of the state machine and bit endiannes which is unclear in the documentation, even with the application note clarifying it [5].

Another thing we found invaluable was the verbosity flag controlled by a different character at each level of abstraction, (similar to what the compilers do in Plan 9) with the lowest printing the MPSSE assembly and the highest printing the ARM context when entering and exiting debug mode. `Jtagfs` can print any of its levels of interaction, which makes it simple to debug new devices and can be interesting for anyone willing to know more about JTAG on the ARM, which has some dark corners and rough edges (specially the bit ordering or the timing of the state machine).

### Experience

While it is slow when reading or writing big amounts of data, mostly because of the roundtrip of the USB protocol, it is still quite usable for regular debugging. It could be made faster by batching together bigger chunks of data or by caching recently accessed data. Both have important drawbacks, considering that reading and writing may have ordering constraints (for example when accessing memory mapped devices), which is why we did not implement them.

`Jtagfs` has showed its power when using it several times. For example, after programming it, we found that just after stopping the core with Plan 9 on it, it would reboot no matter what we did. After some poking and probing, we learnt that it was the watchdog device rebooting the machine when the processor was stopped. Just by writing some acid code, we were able to to disable and reenable the watchdog as needed. Another interesting experience was debugging some code for traps that had failed to work for a long time and we did not understand why. It turned out that in the end we were using an instruction which was not supported in the machine, but what had stalled us for days was debugged in a couple of hours using `jtagfs`.

The most important feature we have missed when using the device are more breakpoint and watchpoint units, which would make debugging simpler, but this is a hardware problem outside of our control. The number of units is also dependent on the core. Software breakpoints in the kernel could be implemented, but with the caches and the pipelines interactions they would probably be quite a feat to get right.

### Related work

There are several programs to interface JTAG providing a backend to gdb, for example OpenOCD [2] or the Blackfin Uclinux gnu toolchain [3]. There are also developer boards and closed software like that of [9]. All of them, or at least the ones we have seen and used, provide at most batch-like capabilities (whereas acid and proc combined provide a

fully programmable interface). Furthermore, the `proc`(3) interface is designed to be portable so it can be easily used from any other programming language and operating system, providing a simple portable interface, whereas the interfaces provided by programs like OpenOCD (OpenOCD provides a gdb commands telnet server) are designed to be used with gdb and not as general.

### Future work

As it is now, `jtagfs` only provides Feroceon support and has only been tried in the Sheevaplug. Support has been added for the Armada, but is untried. Most of the software should work without modification on any ARM 7 or 9 as it has been written to be very portable. To support other boards the id code for the processor needs to be added and the configuration necessary to deal with the wiring of the board.

After the Sheevaplug has been running for a short period of time, the JTAG interface stops responding unless it has already been accessed, though this looks like it is a characteristic of the hardware and the same happens to OpenOCD on Linux. In any case, when the JTAG does not respond, it is detected in the identification phase and it can still be reset through the JTAG interface. As long as there is some interaction with the JTAG (it can be only to identify it) early in the boot process, the JTAG works flawlessly.

There are other capabilities of the ARM chips which can be accessed from the JTAG and which could be interesting. One of them is the Embedded Trace Macrocell or ETM [8] an instruction and data tracing interface to the processor. Another interesting capability is the DCC or Debug Communications Channel. It provides three registers to access a bidirectional serial communications channel (polled or interrupt driven) for printing and debugging using the JTAG. From inside the processor, the target sees the DCC as the coprocessor 14 using MCR and MRC. From the JTAG these registers can be accessed by means of scan chain 2.

Another interesting capability that could be implemented is to freeze the processor from within the kernel, by setting the debug registers. Then, the hardware could be accessed from the jtag port using `jtagfs`. We have not done this, but it should be trivial to do, as it is just setting a register. One good place to do this, for example, would be in the panic routine, so that when a kernel panics, it can be inspected.

The JTAG interface could be also used to inject a loader or a kernel as a last resort for a bricked device or to read or write the contents of the flash.

Last but not least, using /proc and acid any software running on the ARM can be debugged. It would be very interesting and not very difficult to add more support for ELF [13] symbol tables and binaries (perhaps using those of plan9ports or go) to Mach. This could enable debugging the Linux kernel or U-boot using acid.

### References


1. F. J. Ballesteros, *Plan 9's Universal Serial Bus*, IWP9, 2009.
2. R. D., Open On-Chip Debugger, *Diploma, Department of Computer Science, University of Applied Sciences Augsburg*, .
3. B. U. http://blackfin.uclinux.org/gf/, Blackfin GNU Toolchain.
4. F. T. D. I. L. http://ftdichip.com, AN2232C-01 Command Processor for MPSSE and MCU Host Bus Emulation Modes.
5. http://infocenter.arm.com, Application note 205 Writing JTAG Sequences for Arm 9 Processors.

6. http://infocenter.arm.com, ARM9E-S Technical Reference Manual.
7. http://infocenter.arm.com, ARM7TDMI-S Core Technical Reference Manual.
8. http://infocenter.arm.com, Embedded Trace Macrocell Architecture Specification.
9. X. L. http://www.xjtag.com, XJTAG company.
10. IEEE, IEEE 1149.1 standard specification, *Standard Test Access Port and Boundary Scan Architecture*, .
11. B. Labs, Plan 9 man pages, *Plan 9 User's manual, Vol 1*, 1995.
12. G. G. Múzquiz, F. J. Ballesteros and E. Soriano, *Usb serial design and experience in Plan 9*, IWP9, 2010.
13. T. I. Standard, Executable and Linking Format (ELF) Specification Version 1.2, *TIS Committee*, .
14. P. Winterbottom, Acid Manual, *Plan 9 Programmer's Manual. AT&T Bell Laboratories. Murray Hill, NJ.*, 1995.

# FTP-like Streams for the 9P File Protocol

*John Floren*
*john@jfloren.net*
February 16, 2011

*ABSTRACT*

The 9P file protocol is used for all file operations in the Plan 9 operating system. Although it is simple and generally effective, 9P tends to show extremely poor performance when used to transfer large files over high-latency links, such as the Internet. This work extends the 9P protocol to introduce the concept of "streams" as used in FTP.

## Introduction

The Plan 9 operating system utilizes a single filesystem protocol, 9P, to access all resources on the system, including devices, the network stack, the windowing system, and the archival backup system. All of these files are made available by file servers, which are mounted to a namespace at the kernel level and then accessed using 9P messages [4].

Although 9P has proven a very effective way to deal with local files and files on local networks, in some ways it is lacking. At its core, 9P operation is synchronous. A message is sent requesting an operation, such as a read, and the client program must then block until the reply comes back from the server. Every read or write request must wait for the entire round-trip time (RTT) between the client and the server before the program can continue. With the rising prominence of the Internet, where latencies of hundreds of milliseconds are common, it has become apparent that accessing and transferring files from distant Plan 9 systems using 9P is extremely slow and inefficient, requiring far more time to complete a file transfer than the more popular Hyper-Text Transfer Protocol (HTTP) or File Transfer Protocol (FTP). In fact, initial tests indicated that over a connection with 50 ms RTT (relatively low for the Internet), transferring a file took nearly four times as long with 9P as it did with HTTP.

Where 9P operates in terms of "chunks" of a file, requested and delivered one piece at a time, HTTP and FTP deal with entire files at once. When a file is requested from an FTP server in passive mode, the FTP server and the client negotiate a separate, new TCP connection to be used to transfer the file all at once. In HTTP, when a GET request is sent over the connection, the entire file is returned immediately (although a specific portion of the file can also be requested, which will also be sent immediately regardless of size) [5]. Because FTP and HTTP push data directly to a TCP connection rather than waiting for it to be requested like 9P, they avoid many of the issues of latency.

This work proposes and implements a method for alleviating the latency problems in 9P. The solution, called "streams" in this text, operates by taking a leaf from FTP's book: it allows the creation of a new TCP connection to transfer file data. These streams, made available to programmers through library functions, can be used in any situation in which sequential reading or writing of a file occurs. Rather than attempting to improve performance with programmer-transpared adjustments behind the scenes, streams extend the POSIX-style file I/O API to put explicit control in the hands of the programmer. As the results show, the introduction of streams to 9P allows Plan 9 to transfer data using 9P just as quickly as with HTTP.

**9P**

Files in Plan 9 are served and accessed using the 9P protocol. Client programs, such as `cp` and `ed`, access files using the familiar `read()`, `write()`, `open()`, `close()`, *etc.* library functions. The C library then translates these function calls into 9P messages, which are sent to the appropriate file server. File servers listen for incoming 9P messages (typically over a network connection or a pipe) and respond to them appropriately. Only one reply is sent per request.

9P messages come in pairs: a client sends a T-message (such as `Tread` ), which is received by the server. The server then responds with an R-message (such as `Rread` ). A T-message will always receive its corresponding R-message, unless an error occurs, in which case the server will respond with the `Rerror` message. The complete set of 9P messages is shown in Table 1.

| | |
|---|---|
| Tversion/Rversion | Exchanges client/server 9P version numbers |
| Tauth/Rauth | Authenticates client with server |
| Rerror | Indicates an error (includes error string) |
| Tflush/Rflush | Aborts a previous request |
| Tattach/Rattach | Establishes a connection to a file tree |
| Twalk/Rwalk | Descends a directory hierarchy |
| Topen/Ropen | Opens a file or directory |
| Tcreate/Rcreate | Creates a file |
| Tread/Rread | Reads data from an open file |
| Twrite/Rwrite | Writes data to an open file |
| Tclunk/Rclunk | Closes an open file |
| Tremove/Rremove | Removes a file |
| Tstat/Rstat | Requests information about a file |
| Twstat/Rwstat | Changes information about a file |

*Table 1. 9P messages [1].*

All connections are initiated with the exchange of a `Tversion/Rversion` pair. In this exchange, the client and server insure that they are both speaking a compatible version of the 9P protocol. At the time of writing, the most recent version was called "9P2000", with an earlier version "9P" long deprecated. If either the client or the server returns a version that the other does not understand, no communication can be performed. Note that the definition allows extensions to be specified by appending characters to the version string following a period [2]; for example, "9P2000.L" represents a version of 9P with extensions for Linux. Anything following the "." is considered optional, thus "9P2000.L" is backwards compatible with "9P2000".

Once the protocol version has been established, a client typically sends a `Tattach` message to attach to a specific file tree. Then, the client is free to `Twalk`, `Tstat`, `Topen`, *etc.* all over the file tree.

When a user opens a file, the client program makes an `open()` system call, which is translated into a 9P message, `Topen`, which is sent to the file server. The file server responds with an `Ropen` message when the file has been opened. The client then sends `Tread` and `Twrite` messages to read and write the file, with the server responding with `Rread` and `Rwrite`. When the transaction is completed, the client sends `Tclunk` and receives `Rclunk` in return, at which point the file is closed.

### Streams

The primary reason for the slow performance of 9P over high-latency links is its T-message/R-message nature. Every read call issued by a program must wait for the entire round-trip time of the connection before getting any information back. This means that programs end up blocking for a very long time while messages are sent across the network.

The addition of "streams" is intended to alleviate this problem. Sequential reading of files is very common; it appears when copying files, playing music and videos, opening a file in an editor, etc. Typically, the programmer knows when a file will be read sequentially, but with the stock 9P implementation that knowledge helps little. Streams, on the other hand, aim to provide programmers with a powerful method for quickly reading in a file sequentially.

The core concept of streams is drawn from the operation of passive FTP. When a client requests a file in passive mode, the FTP server and the client negotiate over the existing connection to set up a new, separate TCP connection for transferring the data [3]. The server then simply writes the entirety of the requested file to the new connection, which the client then reads.

Typically, all 9P messages between a given client computer and its server are multiplexed over a single TCP connection. This means that multiple programs may all be sending messages over a single connection. To initialize a stream, a client program sends a `Tstream` message to the server, requesting a new stream. The server in turn responds with a `Rstream` message indicating an IP and port which represent a new, dedicated TCP connection for transferring a file.

### Library Interface

Streams are made available to the programmer through the C library, `libc.` There are two ways in which programmers may use streams: in a regular (client) program, or in a file server.

Regular programs such as `cp` or an MP3 player utilize streams using the library functions `stream, sread, swrite,` and routines`sclose.` These

| Stream* stream(int fd, vlong off, char isread) |
|---|
| long sread(Stream* s, void *buf, long n) |
| long swrite(Stream* s, void *buf, long n) |
| int sclose(Stream* s) |

*Table 2. libc functions.*

```
typedef struct Stream {
  int ofd;            // The underlying file being streamed
  int conn;           // The TCP connection
  char *addr;         // The server's IP and port
  vlong offset;       // Current offset into the file
  char isread;        // Read/write flag
  char compatibility; // Compatibility mode flag
} Stream;
```

*The Stream structure*

The operation of the functions is very similar to the corresponding functions for regular reading and writing. The `stream` function initializes a stream given an already-open file descriptor. It is at this point that the directionality of a stream (read or write) is determined. The `sread, swrite,` and `sclose` functions all behave just like the corresponding `read, write,` and `close` functions already existing in libc.

Since some servers may not support 9P streams, if the creation of a stream fails the functions `sread, swrite,` and `sclose` work in "compatibility" mode, issuing regular reads and writes on the original file

descriptor.

### Streaming 9P Messages

9P servers and the kernel drivers deal with 9P at the level of 9P messages, rather than the POSIX-like abstractions seen by regular client programs. To implement streaming, two new 9P messages were added, `Tstream` and `Rstream,` which are formatted as shown in Table 3. Numbers in brackets represent field sizes in bytes.

| size[4] Tstream tag[2] fid[4] isread[1] offset[8] |
|---|
| size[4] Rstream tag[2] count[4] data[count] |

*Table 3. Format of streaming 9P messages.*

Each message consists of a string of bytes, with the size defined by the size element at the beginning. The tag field is common to both messages and is used by the client to identify the messages as belonging to the same conversation.

The Tstream message specifies an fid, a flag for reading or writing, and an offset into the specified file. An fid is a 32-bit integer used to identify a specific active file on the file server and is in many ways analogous to a file descriptor in a user program.

The Rstream message is returned by the file server and contains a network address string in the data field, the length of which is defined in the count field. The network address is in the format "tcp!x.x.x.x!yyyyy", where x.x.x.x is the server's IP and yyyyy is a TCP port on the server. This network address is used by the client to connect to the file server, creating a TCP stream over which file data is sent.

### Server-Side Design

Every 9P server program must be modified to support streaming explicitly. When the 9P version is negotiated at the beginning of a session, a streaming-compatible server will report its version as "9P2000.s" rather than the default "9P2000". As the definition of the `Tstream/Rstream` messages states, anything in a version string following a period represents an optional extension to 9P. Thus, a streaming server is still fully compatible with a client which does not handle streaming, and vice versa.

In general, to add streaming to a 9P server it is necessary to add a new handler for the `Tstream` message type. When such a message comes in, this handler (forked off as a new process to avoid blocking) must begin listening on a new TCP port, then send the server's IP and the port number back in an `Rstream` message. When the client connects to that port, the server then begins reading/writing file data to/from the connection until either the file is fully sent (in the case of a read stream) or the connection is closed (in the case of a write stream).

### Implementation

The implementation of streams in 9P introduced changes in essentially every level of the system. User-level functions were introduced into the C library and user programs were modified to use those functions. A new system call was also added, and streaming support was included in device-specific drivers.

At the lowest level is the driver support. It is necessary for each kernel driver to know how to set up a stream; a specific function was added to each to request and create a new stream. However, all remote filesystems are mounted through the `devmnt` device; the upshot of this is that streams only had to be fully implemented in `devmnt,` while other devices could simply have their stream functions return failure, forcing the stream into compatibility mode.

To act as an interface between user-space programs and the device drivers mentioned above, a `pstream` system call was added. This system call is called when a new stream is requested by a user-level program. It does little more than parse its arguments, convert the given file descriptor to an in-kernel channel, and

then call the appropriate device-specific function based on the location of the requested file. If streaming is not supported, `pstream` returns -1.

The C library functions mentioned above were added to the `/sys/src/libc/9sys` directory. In general, these functions were quite simple to implement. The `stream` function simply creates a new `Stream` structure, sets some options, and calls the `pstream` system call, which returns an IP address and port. The function then calls `dial` to open a new TCP connection and returns to the user program. If the remote server or the device driver is not streaming compatible, `pstream` will have returned -1, a compatibility flag is set, and no TCP connection is created.

The `sread, swrite,` and `sclose` functions behave "appropriately" based on the status of the stream's compatibility flag.

To summarize, a user program calls `stream` on an open file descriptor to obtain a new stream.

**Tests and Results**

In order to test streaming 9P, two programs were modified to utilize streaming. The `cp` user program and the `exportfs` 9P server were, over the course of about an hour, modified to support streams. In the case of `cp,` the changes totalled about 4 lines; the modified (streaming) version of `cp` is here referred to as `scp.` Modifications to exportfs were slightly more extensive, but basically amounted to making the server report its version as "9P2000.s" and adding an 80-line `Tstream` handler function.

A network was set up for simulating the latency of the Internet. As Figure 1 shows, a client machine ( `illiac` , booting standalone from a local disk) and a server ( `p9` , acting as a combined CPU/auth/file server), both running Plan 9, were separated by a gateway computer ( `sigil` ) running Linux. This gateway was configured to use the `netem` kernel extensions, which allow the introduction of arbitrary latencies for network packets passing through the gateway.

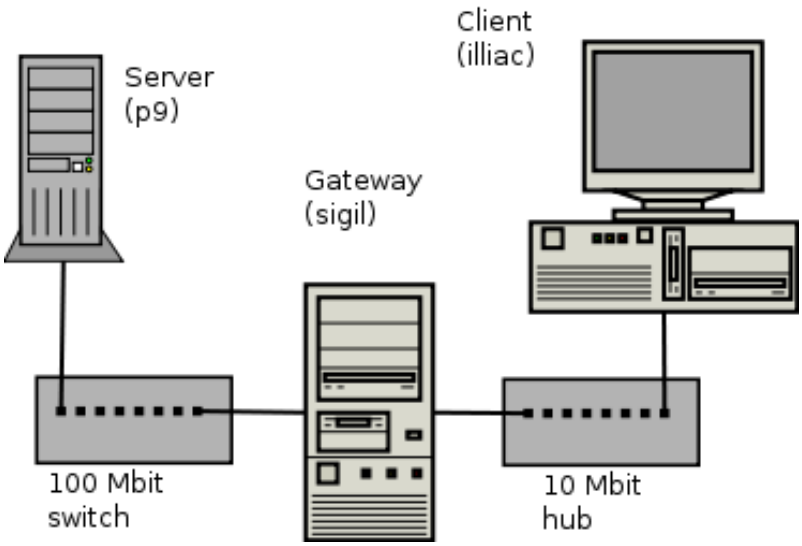



*Figure 1. Testing network*

A set of tests were devised to find the speeds at which files could be transferred using 9P, HTTP, and streaming 9P. 10 MB, 50 MB, 100 MB, and 200 M B files were randomly generated, then transferred over the network via the three different protocols with round-trip latencies of 500 µs, 15 ms, and 50 ms.

The server was configured to run the modified (streams-enabled) `exportfs` server and Plan 9's HTTP server. On the client, the server's namespace was mounted locally using the command `import p9 / /n/p9` , which connected to the server's running `exportfs` process.

In order to transfer a file from the server to the client using streams, commands of the form `scp /n/p9/usr/john/random10M /dev/null` were issued, using the specially modified streaming `cp` program to copy the files. Files were transferred using non-streaming 9P with commands such as `cp /n/p9/usr/john/random10M /dev/null`, and files were transferred via HTTP with the command line `hget http://p9/random10M > /dev/null time` command (also present in Unix), which reports the time required to complete a process. Table 4 and Figure 2 show the resulting transfer times when no artificial latency was induced, giving an average round-trip time of 500 µs. At such a low latency, the performance of HTTP, 9P, and streaming 9P are essentially identical.

| | **9P (sec.)** | | **HTTP (sec.)** | | **Streaming 9P (sec.)** | |
|---|---|---|---|---|---|---|
| **File Size (MB)** | **mean** | **std. dev** | **mean** | **std. dev** | **mean** | **std. dev** |
| 10 | 10.91 | 0.51 | 12.66 | 0.37 | 12.71 | 0.31 |
| 50 | 59.21 | 2.34 | 62.75 | 0.33 | 62.11 | 0.29 |
| 100 | 126.96 | 5.72 | 125.30 | 0.58 | 125.58 | 0.58 |
| 200 | 262.40 | 0.37 | 251.41 | 0.59 | 251.53 | 0.87 |

*Table 4. HTTP vs. 9P vs. Streaming 9P, no induced latency, average RTT 500 µs*

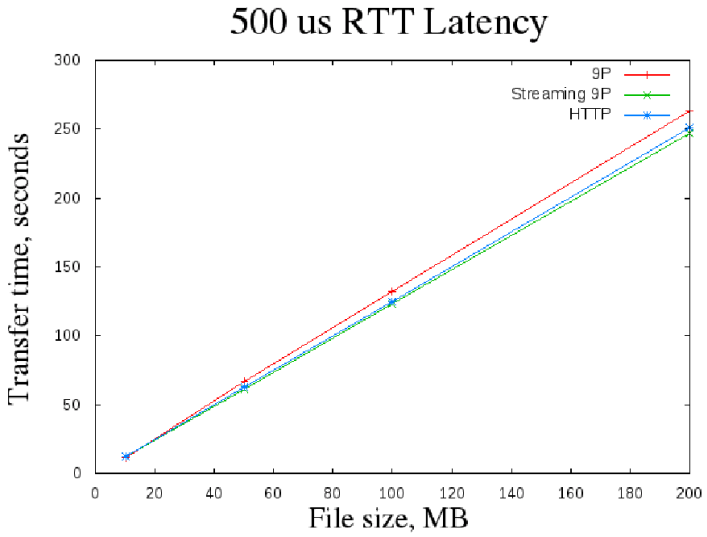



*Figure 2. HTTP vs. 9P vs. Streaming 9P, no induced latency, average RTT 500 µs*

When the latency was increased to 15 ms round-trip, as shown in Table 5 and Figure 3, regular 9P immediately fell significantly behind HTTP, while streaming 9P maintained almost exactly the same performance as HTTP. Differences between the transfer speeds for the two protocols were small enough to be considered simple experimental variance.

| | 9P (sec.) | | HTTP (sec.) | | Streaming 9P (sec.) | |
|---|---|---|---|---|---|---|
| **File Size (MB)** | **mean** | **std. dev** | **mean** | **std. dev** | **mean** | **std. dev** |
| 10 | 30.85 | 1.34 | 14.47 | 0.55 | 14.41 | 0.11 |
| 50 | 156.29 | 2.84 | 71.41 | 0.41 | 70.91 | 0.51 |
| 100 | 319.88 | 5.8 | 144.44 | 0.50 | 142.10 | 1.07 |
| 200 | 647.22 | 1.87 | 286.56 | 2.06 | 284.60 | 0.61 |

*Table 5. HTTP vs. 9P vs. Streaming 9P, no induced latency, average RTT 15 ms*

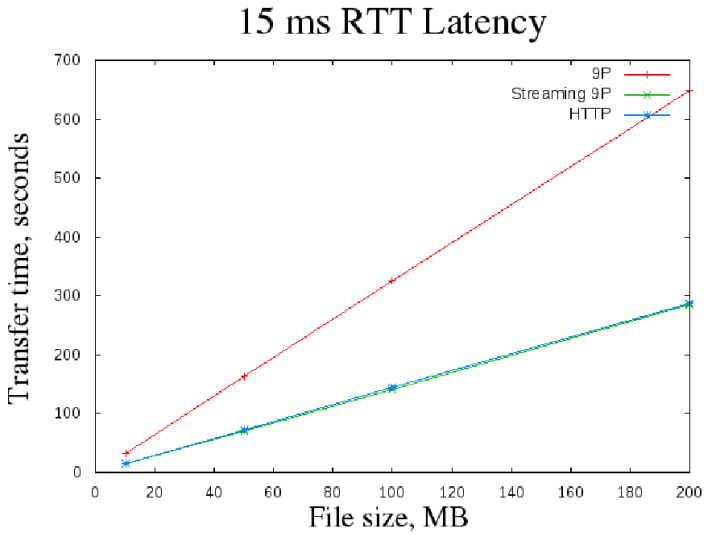



*Figure 3. HTTP vs. 9P vs. Streaming 9P, no induced latency, average RTT 15 ms*

Figure 4 and Table 6 show the results at 50 ms induced latency. As in the previous test, HTTP and streaming 9P took almost exactly the same amount of time to transfer the same data, remaining within a standard deviation of each other, while regular 9P fell even more behind.

| | 9P (sec.) | | HTTP (sec.) | | Streaming 9P (sec.) | |
|---|---|---|---|---|---|---|
| **File Size (MB)** | **mean** | **std. dev** | **mean** | **std. dev** | **mean** | **std. dev** |
| 10 | 75.15 | 0.41 | 19.58 | 0.23 | 19.88 | 0.12 |
| 50 | 379.67 | 2.04 | 96.71 | 0.46 | 96.83 | 0.66 |
| 100 | 767.93 | 5.19 | 193.10 | 1.40 | 193.23 | 0.58 |
| 200 | 1543.12 | 0.62 | 385.292 | 1.85 | 386.02 | 1.77 |

*Table 6. HTTP vs. 9P vs. Streaming 9P, no induced latency, average RTT 50 ms*

The final test that was performed checked the concurrency capabilities of streaming 9P compared to HTTP and regular 9P. The commands that had been used previously, for example `time cp /n/p9/usr/john/random10M /dev/null,` were followed by an `&` character, which caused the command to be executed in the background and allowed for the execution of multiple client programs at the same time. The Plan 9 OS has a feature which allows a user to enter multiple commands while in "hold" mode; upon exiting "hold" mode, the shell begins processing all the commands one after another. This allowed multiple clients to be launched almost simultaneously.

Tests were performed using two, four, and eight simultaneous client programs on the client PC. For the

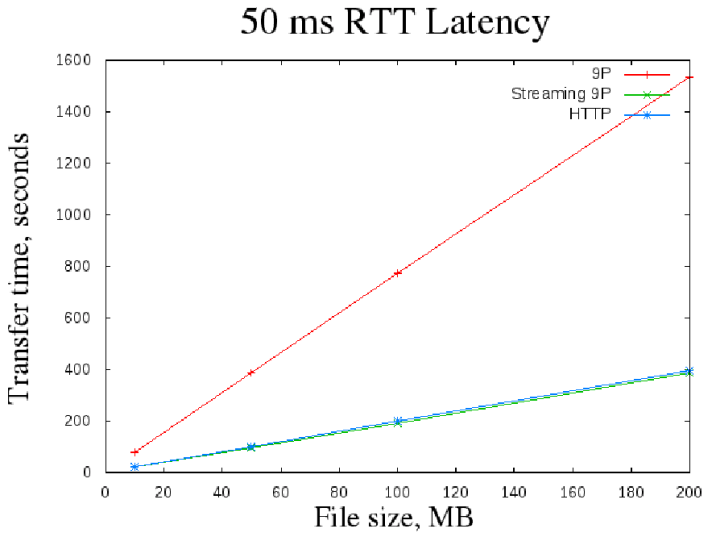



*Figure 4. HTTP vs. 9P vs. Streaming 9P, no induced latency, average RTT 50 ms*

test, a round-trip latency time of 50 milliseconds was used with a file size of 10 MB. The execution times for each instance of the programs were averaged to get the results shown in Table 7 and Figure 5.

| Number of Clients | Streaming 9P (sec.) | HTTP (sec.) | 9P (sec.) |
|---|---|---|---|
| 2 | 30.96 | 28.51 | 79.75 |
| 4 | 53.28 | 52.53 | 100.81 |
| 8 | 104.24 | 101.93 | 158.53 |

*Table 7. 10 MB file transfer speeds with multiple concurrent clients, 50 ms RTT*

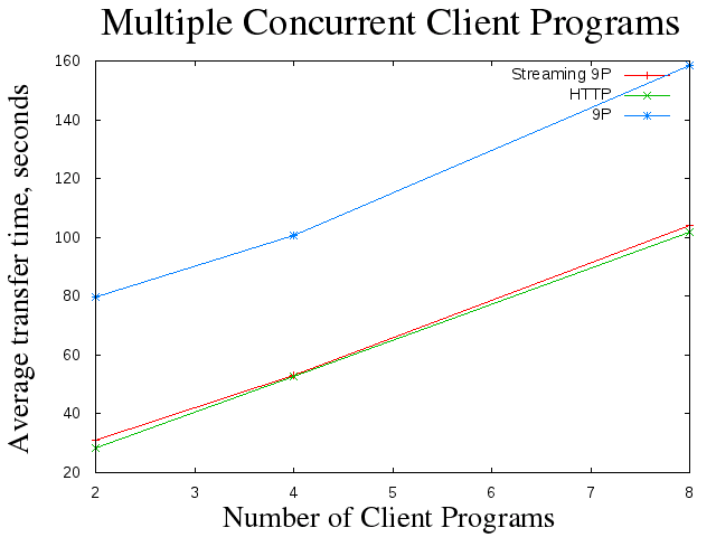



*Figure 5. 10 MB file transfer speeds with multiple concurrent clients, 50 ms RTT*

As the results show, all three protocols experienced increased transfer times as the number of concurrent transfers increased. Streaming 9P and HTTP both showed approximately the same level of scalability

within the limited confines of this test, with performance decreasing in a slow and linear fashion.

**Summary**

The addition of streams functionality provides one way in which 9P's latency problems may be overcome. By allowing programmers to explicitly indicate that they intend to read or write a file sequentially, streaming functionality is used only where appropriate.

As the results show, the use of streams allows 9P to transfer files at a rate comparable to that of HTTP. The modifications necessary to enable streaming in a client program are not onerous and can potentially increase the speed of file operations many times over.

Future work would include the conversion of more user programs and servers to use streams. Any file server which listens for connections over the network could benefit from the inclusion of streaming capability while still remaining backward-compatible with non-streaming clients. Specifically, the disk-backed file servers Fossil and Venti are ideal targets for streaming. When a Plan 9 system such as a terminal mounts a root file system, it typically connects to a remote Fossil server; providing streaming within Fossil would allow users to experience more efficient file access for almost all of their files. On the client side, the image and document viewers could benefit from streaming file access, as would the MP3 decoder; all of these programs regularly access large files sequentially.

**Bibliography**

[1] *Introduction to the Plan 9 file protocol, 9P*. Plan 9 online manual, section 5. http://plan9.bell-labs.com/magic/man2html/5/0intro.

[2] *Version*. Plan 9 online manual, section 5. http://plan9.bell-labs.com/magic/man2html/5/version.

[3] J. Postel and J. Reynolds. RFC 959: File transfer protocol, October 1985.

[4] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. 8(3):221 254, Summer 1995.

# To Stream or not to Stream

*Jeffrey Sickel*

Corpus Callosum Corporation
Evanston, IL 60202 USA
jas@corpus-callosum.com

*ABSTRACT*

This paper describes a technique to manage multiple serial devices that switch data transmission modes between request-response and streaming protocols. It utilizes the ideas of coroutines and communicating sequential processes to build concurrent input and output processing routines for each device. The example program leverages Limbo's buffered channels to concurrently queue and process data from multiple inputs in soft real-time.

## 1. Introduction

Scores of researchers have promoted various structured programming techniques to manage concurrency over the years. Conway [1], Dijkstra [2], Hoare [3,4], and Kahn and MacQueen [5] have all contributed to the body of knowledge that frames how concurrent programs are written today. Yet, even with this history, research, and practice of programming, the processing of input and output (I/O) continues to be reduced to routines that must run to completion before another task can continue. This inherent single-tasked nature of handling I/O means that user programs, especially with graphical front ends, need to investigate concurrent approaches to managing multiple I/O interfaces. For example, reading data from a slow remote device while at the same time needing to service other input tasks requires managing system state successfully to prevent blocking conditions from interrupting the logic flow.

Coroutines, coined by Conway and expanded by later researchers, are a proven way to handle various issues surrounding the logical segmentation of code. The definition, as published by Conway [1:396], elucidated *separability,* or modularity:

> [The coroutine] may be coded as an autonomous program which communicates with adjacent modules as if they were input or output subroutines. Thus, coroutines are subroutines all at the same level, each acting as if it were the master program when in fact there is no master program.

Leveraging this modularity helps in designing optimal input- and output-handling routines based on machine or process state at any given time. Though the logic is modularized, state changes stored by the coroutines do not fully model a concurrent or parallel system as the join required for two coroutines to exchange data will cause at least one to wait, or block, until the other is able to send or receive data to the peer (see Knuth [6] for examples).

In order to achieve greater concurrency a mechanism is required to leverage the I/O handling of coroutines into a more approachable framework. The seminal paper by Hoare [3], *Communicating Sequential Processes* (CSP), provides such a framework and influenced the design of the Limbo programming language. CSP is the mechanism used by Limbo channels to provide bi-directional communication between processes. Limbo processes are able to run concurrently, in parallel if the underlying hardware supports it, by leveraging constructs provided by CSP. Ritchie [7] describes the simplified use of Limbo channels to handle reading data from a single device. Updates to Inferno and Limbo since Ritchie's document now include buffered channels, a

language extension that allows for up to *n* buffer size of values to be sent without blocking. This paper contributes to the available documentation on using Limbo channels to manage concurrent I/O tasks.

The final development stage of a new aero-acoustic levitator (AAL) [8] required a program to interface, control, and display data from various serial devices. Some of these devices use a simple request-response protocol, whereas others switch from request-response communication to a streaming protocol. All of the devices operate in modes where sending a request and then blocking to wait for a response will not suffice, as there are certain conditions where a message will be sent from the device out of order from a sequential request-response loop initiated by the user control program. In order to manage these message states, this implementation uses multiple asynchronous processes as coroutines to handle all of the data management and machine control. The user is presented with a seamless interface isolated from the underlying serial communication tasks.

The following sections present *aal/pyro* (Figure 1), a Limbo program that uses CSP programming techniques to manage the I/O from multiple devices. The system combines a remote pyrometer with a linear XY translator, three serial devices in total. The program plots, and optionally logs, the stream of temperature measurements sent from the pyrometer. Position of the pyrometer is controlled using the XY translators through numeric key entry and a graphical view that accepts coordinates converted from mouse input. These tasks are accomplished by building program structures to handle I/O from the various serial interfaces. The implementation leverages Limbo's buffered channels for inter-process communication to control parallel data feeds.

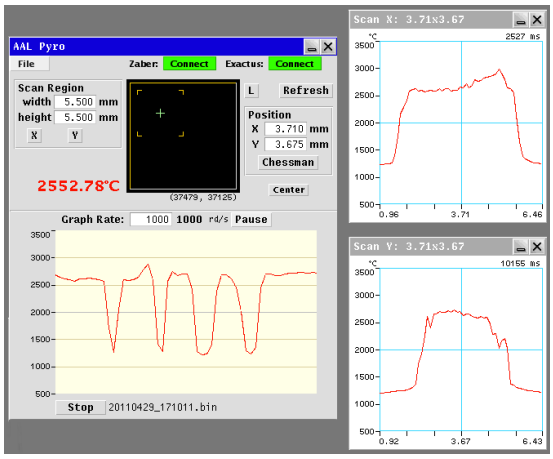



**Figure 1** aal/pyro application

## 2. Design

The *aal/pyro* program is a graphics interface that manages I/O from multiple physical devices. Different serial communications protocols are used to process data from each device. Blocking conditions common in I/O routines are eliminated from the control flow by separating each task into independent processes (thought of as coroutines). The program manages these coroutines by communicating through Limbo channels. By isolating the blocking functions into separate coroutines, the program is able to run seamlessly without interrupting the interactive program flow. The Limbo *alt* (alteration) statement [9] is used to manage concurrent communications between each of the key coroutines that depend on data processing. The *alt* is like a *case* or *switch* statement but specific for handling multiple communications channels. Thus the program logic can be declared in simple synchronous terms even while handling asynchronous events.

On startup, the *pyro* process (Figure 2) spawns off a single *timer* process as a mechanism to

signal the *pyro* process to flush any out-of-sync communications from the XY translator (Zaber). The *timer* does nothing until a connection is made to remote devices. The main *alt* event loop in the *pyro* process handles user interaction and drawing routines representing the coordinate space of the Zaber devices. Plotting and other drawing updates are managed using separate processes started after connection to the Exactus pyrometer is made.

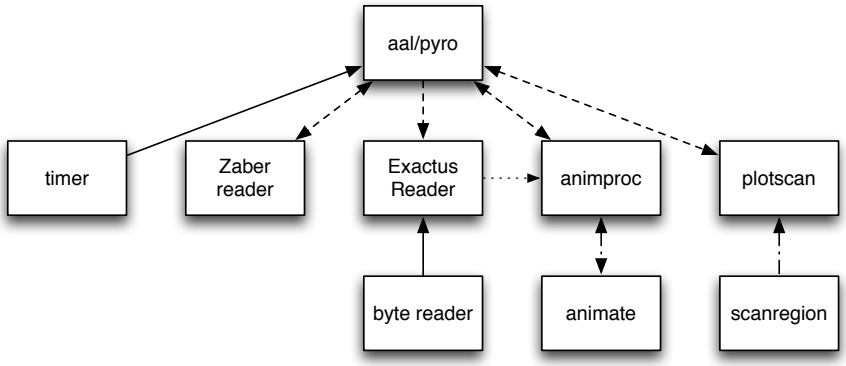



**Figure 2** aal/pyro processes

All serial devices have a minimal data structure referred to as a *port*. For simplicity, each port can be opened from a serial device file using *sys->open()* or via *sys->dial()* to connect to a remote service. The connection routine is itself spawned off from the *pyro* process so that the blocking *open()* or *dial()* calls do not inhibit other user interface interaction. A message will be sent over a channel to the *pyro* process notifying whether a successful connection was made. A valid connection will enable additional UI elements and spawn a dedicated process to manage reading from the device.

Connecting to the Exactus pyrometer starts up a series of other processes. The first is the reader used to analyze the different input data from the device. The reader spawns off a blocking byte reader that simply receives all data from the pyrometer serial interface and sends each byte back over a buffered channel, decoupling the analysis from the physical device. A separate process group is used for on-screen drawing. The *animproc* process is spawned after connection to the pyrometer and plots temperature data using a coordinated animate process. A separate *plotscan* process is created by a user-initiated event to plot temperature data in relation to the axial position of the pyrometer. Each of the interfaces is declared in its own module as defined in the following sections.

### 2.1. Zaber

The Zaber XY translators are interfaced via a single RS232 link that communicates with both the X and the Y linear stage. The serial protocol provides a clean, fixed byte-length message format for both sending and receiving data. Though the protocol works like a sequential transmit-response operation, not all commands to the device return a response message. Additionally, there is no guaranteed order to the responses and there are certain cases where the device issues state data interleaved between other response messages.

The connection to the device creates an instance of a Zaber port:

```
Port: adt
{
    pid:    int;
    local:  string;
    ctl:    ref Sys->FD;
    data:   ref Sys->FD;
    rdlock: ref Lock->Semaphore;
```

```
        wrlock: ref Lock->Semaphore;
        buf:    array of byte;

        write:  fn(c: self ref Port, b: array of byte): int;
    };
```

After successfully connecting to and opening the device, the *pid* is set to the process id of the spawned off *reader()*. A buffer of received bytes is stored in the *port* structure. The semaphore *rdlock* is used as the bytes buffered in the Zaber *reader* process are validated and consumed in the *pyro* process.

Spawning the *reader* off into its own process allows the blocking *sys->read()* call to continually read data without interfering with the rest of the application event management. Data received from the device is stored in a buffer until a subsequent routine polls the buffered data.

```
reader(p: ref Port, pidc: chan of int)
{
    pidc <-= sys->pctl(0, nil);

    buf := array[1] of byte;
    for(;;) {
        while((n := sys->read(p.data, buf, len buf)) > 0) {
            p.rdlock.obtain();
            if(len p.avail < Sys->ATOMICIO) {
                na := array[len p.avail + n] of byte;
                na[0:] = p.avail[0:];
                na[len p.avail:] = buf[0:n];
                p.avail = na;
            }
            p.rdlock.release();
        }
        # error, attempt reconnect and try again
        openport(p);
    }
}
```

The parent *pyro* process checks for responses using ad hoc interleaves with directed timeouts. This is accomplished by calling *readreply()* whenever the parent processes needs to poll data from the device. Each *readreply()* call passes a millisecond time out parameter to enable logic flow to continue if a response has not be received.

The reply may be *nil* due to a timeout. If the response is not returned, it does not matter to the rest of the system state as no dependency is based on actual responses unless specifically requested in another routine. Scanning processes, discussed later, require exact position details and thus explicitly wait until the proper return has been received.

```
readreply(p: ref Port, ms: int): ref Instruction
{
    if(p == nil) return nil;
    if(ms < 0) ms = 60000;      # arbitrary maximum of 60s

    r : ref Instruction;
    for(start := sys->millisec(); sys->millisec() <= start+ms;) {
        a := getreply(p, 1);
        if(len a == 0) {
            sys->sleep(1);
            continue;
        }
        return a[0];
    }

    return r;
}
```

*Readreply()* calls the function *getreply()* to scan the buffer and check for valid instructions.

```
getreply(p: ref Port, n: int): array of ref Instruction
{
    if(p==nil || n <= 0)
        return nil;

    b : array of byte;
    p.rdlock.obtain();
    if(len p.avail >= 6) {
        if((n*6) > len p.avail)
            n = len p.avail / 6;
        b = p.avail[0:(n*6)];
        p.avail = p.avail[(n*6):];
    }
    p.rdlock.release();

    a : array of ref Instruction;
    if(len b) {
        a = array[n] of { * => ref Instruction};
        for(j:=0; j<n; j++) {
            i := a[j];
            i.id = int(b[(j*6)]);
            i.cmd = int(b[(j*6)+1]);
            i.data = b[(j*6)+2:(j*6)+6];
        }
    }
    return a;
}
```

Access to the *port* available data buffer is coordinated between the Zaber *reader()* process and any calls to the *getreply()* function through the semaphore *p.rdlock.* The semaphore is used to insure that the byte reading and message interpretation do not conflict while accessing the buffer. The *getreply()* routine obtains the semaphore lock, then checks the length to determine if a valid message is contained in the data. If the buffer contains enough bytes for *n* messages, it will store those bytes and trim the buffer. The semaphore is then released to allow the Zaber *reader* process to continue filling the buffer as new bytes are received from the device. This coordinated hand-off using the *readreply()* loop means that timeouts can be used while polling for new messages from the XY translators without blocking the *pyro* process.

### 2.2. Exactus

Making the Exactus pyrometer control transparent to the end user is the primary task of the *pyro* program. The device is connected through an RS232 or RS422 interface. The AAL connects to the pyrometer over TCP/IP using a Perle IOLAN SDS as a proxy for transferring the raw bytes to and from the device. The Exactus uses two serial protocols for normal operation: a request-response protocol known as Modbus, and a raw byte stream of data termed the legacy Exactus mode. When in the streaming mode, it is capable of transmitting up to one measurement per millisecond. Though this rate is not fast by modern computing standards, the mode switching over the same serial interface defines the way we have to implement the byte reader.

The pyrometer streaming mode leverages buffered channels to send decoded data frames to the *animproc* process that is dedicated to converting those data into numerical forms presented in a graphical view and optionally to a log file. On the one hand, this is no different than storing the data in a buffer and having another process continually poll for new entries in the same way one would let the request-response loop manage synchronous communications. But in this case the reader process isolates the mode switching and continually monitors the input bytes for proper message structure for both states. The resulting data streams are typed as they are read and subsequently handed off to the respective end points.

### 2.2.1. Modbus

On startup, the pyrometer communicates over its serial link using the Modbus RTU protocol. Modbus is a communications protocol used by many industrial devices and comes in three implementations: *RTU, ASCII,* and *TCP/IP.* The Exactus uses a subset of the RTU protocol to read and write coil and register values on the device. Each message is framed by 3.5-character times of silence (at 115.2kbps, a 303.819µs break between messages). All messages must be initiated by the *aal/pyro* program, as it is the master node in the serial configuration. The primary user interaction that utilizes the Modbus mode is to change the sampling rate, known as graph rate, and switch back into the Exactus streaming mode during actual data collection.

The Limbo Modbus module has been modeled on the 9P or Styx modules due to its transmit and response message structure. There are 19 function codes and an error type declared within the data structure. Unlike the incremental pairing of the T and R types in 9P, Modbus uses the same function code values when declaring both the transmit and response structures. For example, the TMmsg structure:

```
TMmsg: adt {
    frame: int;
    addr: int;              # 1 or 2 bytes
    check: int;             # 0 or 2 bytes
    pick {
    Readerror =>
        error: string;
    Error =>
        fcode: byte;
        ecode: byte;
    Readcoils =>
        offset: int;        # 2 bytes, 0x0000 to 0xFFFF
        quantity: int;      # 2 bytes, 0x0001 to 0x07D0
    Readdiscreteinputs =>
        offset: int;
        quantity: int;
    Readholdingregisters =>
        offset: int;
        quantity: int;      # 2 bytes, 0x0001 to 0x007D
…
    read:   fn(fd: ref Sys->FD, msglim: int): ref TMmsg;
    packedsize: fn(nil: self ref TMmsg): int;
    pack:   fn(nil: self ref TMmsg): array of byte;
    unpack: fn(b: array of byte, h: int): (int, ref TMmsg);
    mtype:  fn(nil: self ref TMmsg): int;
};
```

is nearly identical to the RMmsg structure for received data. As the TMmsg is requesting data from a certain register or coil, the return RMmsg message would include the actual results, as in:

```
    Readholdingregisters =>
        count: int;
        data: array of byte;    # registers, N (of N/2 words)
```

After a process writes a TMmsg, it will block waiting for the reply by calling *readreply()*:

```
EPort.readreply(p: self ref EPort, ms: int): (ref ERmsg, array of byte, string)
{
    if(p == nil)
        return (nil, nil, "No valid port");

    limit := 60000;         # arbitrary maximum of 60s
    r : ref ERmsg;
    b : array of byte;
    err : string;
    for(start := sys->millisec(); sys->millisec() <= start+ms;) {
        (r, b, err) = p.getreply();
        if(r == nil) {
```

```
            if(limit--) {
                sys->sleep(5);
                continue;
            }
            break;
        } else
            break;
    }

    return (r, b, err);
}
```

Note the similarity to the *readreply()* used to access the Zaber devices. Both of these use one function in a loop to poll the buffer through the *getreply()* function. By doing so, the developer can schedule events as needed and continue execution with simple recovery in the case of an error.

### 2.2.2. Streaming

The Exactus legacy streaming mode is a setting where the device sends out a continuous stream of messages at a set rate. The four message types are *temperature, current, dual* (temperature and current), and internal *device* temperatures. The stream contains packets of data messages of variable length that consist of a header byte defining the type, followed by one or more 32-bit IEEE 754 binary floating point values. The portion of the packet encompassing the floating point bytes may include escape codes masking the type and other reserved bytes within the message, thus creating a variable length packet.

The *temperature* and *current* types have a variable packet size of 5-9 bytes, determined by the escape sequences used in packing the message. The *dual* and *device* types packet size is 9-17 bytes in length. The lack of a length attribute in the Exactus message protocol mandates that each byte received be scanned and evaluated to test for message completion. The data structure used in Limbo to represent an Exactus message is:

```
Emsg: adt {
    pick {
    Temperature =>
        degrees:    real;
    Current =>
        amps:       real;
    Dual =>
        degrees:    real;
        amps:       real;
    Device =>
        edegrees:   real;
        cdegrees:   real;
    Version =>
        mode:       byte;
        appid:      byte;
        vermajor:   int;
        verminor:   int;
        build:      int;
    Acknowledge =>
        c:      byte;
    }

    unpack: fn(b: array of byte): (int, ref Emsg);

    temperature:    fn(m: self ref Emsg): real;
    current:    fn(m: self ref Emsg): real;
    dual:       fn(m: self ref Emsg): (real, real);
    device:     fn(m: self ref Emsg): (real, real);
    acknowledge:    fn(m: self ref Emsg): byte;
    text:   fn(m: self ref Emsg): string;
};
```

There is no hinting for the sampling time from the device; the receiver must calculate the timing interval based on the receipt of bytes. The timing accuracy depends not only on the resolution of *sys->millisec()* but also on any latency in the receipt of the bytes from the device. Though latency can be an issue at higher transmission rates, the maximum 1kHz sample rate from the pyrometer is successfully handled.

### 2.2.3. Byte stream processing

Switching states between Modbus and Exactus modes requires a slightly more complex process structure for validating bytes received from the device than defined for the Zaber interface. *Aal/pyro* creates a reference *EPort* to store all of the connection elements:

```
EPort: adt
{
    mode:   int;                # Exactus or Modbus
    maddr:  int;                # Modbus address
    temp:   real;               # Last measured temperature
    rate:   int;                # Graph rate
    path:   string;
    ctl:    ref Sys->FD;
    data:   ref Sys->FD;
    wdata:  ref Sys->FD;
    rdlock: ref Lock->Semaphore;
    wrlock: ref Lock->Semaphore;
    buffer: array of byte;      # bytes from reader
    pids:   list of int;
    tchan:  chan of ref Exactus->Trecord;
    ms: int;                    # ms start of last packet

    write:     fn(p: self ref EPort, b: array of byte): int;
    getreply:  fn(p: self ref EPort): (ref ERmsg, array of byte, string);
    readreply: fn(p: self ref EPort, ms: int):
            (ref ERmsg, array of byte, string);
};
```

When a process is spawned off to connect to the device, the *path* is stored and *sys->dial()* is called. After successfully establishing a connection, the members *ctl, data,* and *wdate* are populated using *sys->open().* The blocking calls *sys->dial()* and *sys->open()* mean that it is important for the main loop to have started this connection routine concurrently as to not block any other elements of the interface or data handling of other I/O components. On successful initialization, the connection process sends a command back over a channel to the *pyro* process and then exits. The notification that the *Eport* has been initialized updates the interface and spawns off the *animproc* process used to plot any data from the pyrometer.

The interesting members of the data structure are the *mode, temp, rate, pids,* and *tchan*, as they are updated based on user interaction controlling the state of the device. Any commands that read or write value changes must first set the device to Modbus mode before making any further requests. The device will be switched back to Exactus streaming mode after the sampling rate is set and data collection is started. The *temp* variable is used as storage and a lookup mechanism for the most recently received temperature message from the pyrometer. *Rate* is a hint field set when the user changes the graphing and sampling rate of the device. The use of *pids* provides a list of subprocess ids to the parent process in case they need to be terminated.

The channel *tchan* is used when the device is in the streaming mode. When *tchan* is not *nil*, then temperature data will be sent from the reading process to another process that acts as a listener. This enables the *animproc* graphing process to be started independently from the *reader.* When a frame of data from the stream is available, it is sent over the *tchan* channel to the *animproc* process for handling within the graphics system.

The two processes that manage the Exactus serial communications are a *reader()* spawned by the *pyro* process, and the blocking *bytereader()*, spawned off by the reader to pick off bytes from the data stream:

```
bytereader(p: ref EPort, c: chan of (int, byte), e: chan of int)
{
    p.pids = sys->pctl(0, nil) :: p.pids;
    buf := array[1] of byte;
    while(sys->read(p.data, buf, len buf) > 0) {
        c <-= (sys->millisec(), buf[0]);
    }
    e <-= 0;
}
```

The channel *chan of (int, byte)* is a buffered channel created in the *reader* process that decouples the blocking reader from the actual decoder used to validate the data stream. The insertion of the *sys->millisec()* in the tuple is used to mark the receipt time of the first byte that begins a message. Latency may offset the accuracy, but it does provide a mechanism to represent time between data messages from the pyrometer.

```
reader(p: ref EPort)
{
  p.pids = sys->pctl(0, nil) :: p.pids;
  c := chan[BUFSZ] of (int, byte);
  e := chan of int;
  spawn bytereader(p, c, e);

  for(;;) alt {
  (ms, b) := <- c =>
    p.rdlock.obtain();
    n := len p.buffer;
    if(n == 0) {
      p.ms = ms;        # used in Trecord, track first received
      l : list of byte;
      if(p.mode == ModeModbus) l = SMBYTES;
      else l = SEBYTES;
      if(!ismember(b, l)) { # frame error
        p.rdlock.release();
        continue;
      }
    }
    na := array[n + 1] of byte;
    if(n) na[0:] = p.buffer[0:n];
    na[n] = b;
    if(p.mode == ModeExactus && p.tchan != nil) {
      (i, m) := Emsg.unpack(na);
      if(m != nil) {
        t := ref Trecord(p.ms, 0.0, 0.0, 0.0, 0.0, 0.0,
                0.0, 0.0, 1.0);
        pick x := m {
        Temperature =>  t.temp0 = p.temp = x.degrees;
        Current =>      t.current1 = x.amps;
        Dual =>
          t.temp0 = p.temp = x.degrees;
          t.current1 = x.amps;
        Device =>
          t.etemp1 = x.edegrees;
          t.etemp2 = x.cdegrees;
        * =>
          t = nil;
        }
        if(t != nil) {
          p.tchan <-= t;
          if(n > i) na = na[i:];
          else na = nil;
        }
      }
    }
    p.buffer = na;
```

```
    p.rdlock.release();
  <-e =>        # bytereader exited, try again
    openport(p);
    spawn bytereader(p, c, e);
  }
}
```

## 3. Graphical interface

The application has two primary graphical components: a representation of the XY position of the pyrometer and a temperature plot of data received. The *pyro* window provides a consolidated interface into the concurrent processes used to coordinate all of the I/O from the attached devices. The user can initiate an additional view that combines the position and temperature data into a consolidated scan plot. By providing a simplified view on top of the coordinated coroutines, the *pyro* process is able to synthesize the translator control and pyrometer data acquisition into a concise view that hides the multiple processes from the user.

### 3.1. Translator control

The Zaber XY translators create a 13x13mm region where the pyrometer can be focused. The data that are returned by the device are in micro-steps and are converted to millimeters for user viewing and numerically entered changes. There is an additional graphical panel that presents the position as a reticle that can be moved by a click in the view. The graphical interaction is managed completely within the *pyro* main alt loop using the *zcmd* channel:

```
c := <-zcmd =>
    if(dflag) sys->fprint(stderr, "zcmd: '%s'0, c);
    if(!plot.lock) {    # max microstep: 131327
        (nil, toks) := sys->tokenize(c, " ");
        pnt := Point(int hd tl toks, int hd tl tl toks);
        ms := real MAXMICROSTEP / real plot.bimg.r.dx();
        x1 := real pnt.x * ms;
        y1 := real pnt.y * ms;
        zsend(Instruction.newwithval(1, Zaber->Cmoveabsolute, int(x1)));
        zsend(Instruction.newwithval(2, Zaber->Cmoveabsolute, int(y1)));
    }
```

The *zcmd* is a string channel named for use within the Tk graphics system. An on-screen Tk panel sends X and Y coordinates over the channel. If the panel *plot* has not been locked by the user to ignore the commands, then the coordinates will be converted into the micro-steps required by the translator and written to the device. The function *zsend()* encodes the instruction into an array of bytes and writes them to the device in order to move to the assigned absolute position. A *zsend()* call is addressed to each translator as they are moved independently. The Zaber devices will not confirm an instruction until after the physical move has completed. The return values from the device may be received out of sequence to the calling convention as the time of travel between positions is the determining factor. There is no requirement to wait for the return result due to the use of a *timer* process checking for new queued return values before updating the display.

The *timer* process sends a message over *tchan* once per second. The *pyro* main alt loop receives the timeout message over the *tchan* and processes the event:

```
<-tchan =>
    if(!scanning) {
        if(zport != nil)
            while((r := zaber->readreply(zport, 1)) != nil)
                processzaber(r);
        if(epid > 0) ecmdc <-= PyroPlot->SAMPLE;
        else if(eport != nil)
            updatedegrees(exactus->temperature(eport));
    }
```

The *scanning* check ensures that the *pyro* process will only poll the Zaber buffer when the *plotscan* process is not actively running. Zaber translator messages are processed before attempting to update an on-screen temperature readout. The channel *ecmdc* is used to message the *animproc* process requesting a new temperature measurement. A response communication would then update the on-screen temperature. If *animproc* is terminated, then the *epid* is set to zero and a blocking call to the pyrometer is made; this requires the main loop to wait for a return from the pyrometer before updating the display and continuing to the next instruction.

### 3.2. Temperature plot

Graphic plotting of temperature data is managed through the *animproc* process spawned after successfully connecting to the Exactus device. Commands controlling logging, sampling rate, and whether or not to plot the data are sent over a channel by the *pyro* process. The buffered channel *recc*, of Exactus Trecord type, is used to receive data processed by the Exactus *reader* while operating in streaming mode:

```
Trecord: adt {
    time:        int;
    temp0:       real;
    temp1:       real;
    temp2:       real;
    current1:    real;
    current2:    real;
    etemp1:      real;
    etemp2:      real;
    emissivity: real;
    pack:   fn(nil: self ref Trecord): array of byte;
    unpack: fn(b: array of byte): (int, ref Trecord);
};
```

The *Trecord* is created by parsing values sent from the Exactus stream data. The *time* field is the milliseconds from the beginning of a log of the data. Logging to disk will use the *pack()* function to create the binary data written out to a filesystem.

Starting *animproc* will in turn spawn off another process to manage the actual drawing routines. This new process, *animate*, receives real values over another buffered channel:

```
animate(top: ref Tk->Toplevel, p: ref Plotter, c: chan of array of real)
{
    for(;;) {
        data := <-c;
        if(!p.paused)
            p.mavg = update(top, p, data);
    }
}
```

All screen drawing is buffered in a fixed-length array of real values before being sent to the *animate* process. The effect of this buffering is to allow the graphical plot to always present at least one minute of historical data. The generated plot point is an average of all the buffered data points; this works well for the full spectrum of graphing rates available from the Exactus pyrometer.

### 3.3. Scanning

During the course of a levitation experiment, it is important to verify that the pyrometer is focused on the sample in order to acquire the best temperature reading possible. In order to accomplish the optimal focusing of the pyrometer, the XY translators are used to scan a region in one dimension while simultaneously collecting temperature and position data. The scanning routine requires all of the prior I/O related functionality in order to accomplish its task in the *pyro* application.

Scanning is handled by spawning off a dedicated *plotscan* process to create and control a new

window where a plot is drawn showing position on the X axis and temperature on the Y axis. The creation of a *plotscan* process sets up the window and spawns off a separate short-lived process to move the XY translator and return temperature data for graphing:

```
c := chan[8] of (int, int, real);
comp := chan of int;
spawn scanregion(rect, c, comp);
```

The *scanregion* process calculates new positions to move the translator and sends command messages in a loop to make the move occur. At each position a temperature measurement is made and the resulting data is sent over a buffered channel, *c*, back to the *plotscan* process where an plot will be rendered on the display. Once the scan completes, a final message will be sent to move the pyrometer back to the starting position. The *scanregion* process will then send a message over the *comp* channel and promptly exit.

All other *aal/pyro* processes continue to run and update their graphical components while the scanning is taking place. Once the *scanregion* process has exited, it is possible for the user to click on the graphic temperature plot to move the pyrometer to a better centered location. The user may then repeat the process to verify optimal pyrometer placement during the experiment.

## 4. Conclusion

System development can be difficult enough without having to worry about I/O blocking a process or threaded programs causing a deadlock. This example detailed how coroutine and CSP models can be used to successfully manage multiple devices by isolating the I/O handlers. The decoupling of the blocking *sys->read()* call, when managed with Limbo channels, can be a useful tool for separating out components of a program to process I/O.

Learning to leverage Limbo channels for inter-process communication may be a foreign idea when coming from other programming languages. Though channels behave like pipes in Unix, the ability to create typed data and easily pass it between processes enables the model to work quite well for concurrent programs. The implementation uses this feature to create independent byte stream readers that gracefully handle serial protocol changes while continually consuming input from external devices.

There are areas where this model could be improved. For one, the reallocation of the buffer array used to store bytes from the input stream can be optimized. Implementing a new data structure to eliminate the semaphore locking could facilitate programming logic simplification. For now, with the constraint of the serial line transmission speeds available to the remote devices, the system performs well enough to capture transmissions from the pyrometer at its maximum rate of 1kHz.

The Limbo source is available upon request from the author.

```
# wc results:

    1103    3215   21536 exactus/exactus.b
     196     578    4055 exactus/exactus.m

    1143    3775   27058 modbus/modbus.b
     248     787    5897 modbus/modbus.m

     424    1276    8449 zaber/zaber.b
     104     248    1944 zaber/zaber.m

    1515    5351   41132 aal/appl/pyro/pyro.b
     365    1035    9056 aal/appl/pyro/pyroplot.b
      29      72     556 aal/module/pyroplot.m

    5127   16337  119683 total
```

## 5. Acknowledgements

The author wishes to thank Belma Hadziselimovic and Omer Hadziselimovic for providing advice on the proper use of the English language, and Jason Bubolz for providing a critical review.

The original work was performed under contract to Physical Property Measurements, Inc. for RWTH Aachen University Institute of Mineral Engineering. Funding for this project was provided by the German Research Association number DFG: INST 222/779-1 FUGG and the Federal state of North Rhine-Westphalia number NRW: 121/4.06.05.08.-566.

## 6. References

[1] Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (July 1963), 396-408. DOI=10.1145/366663.366704 http://doi.acm.org/10.1145/366663.366704

[2] E. W. Dijkstra. 1965. Solution of a problem in concurrent programming control. *Commun. ACM* 8, 9 (September 1965), 569-. DOI=10.1145/365559.365617 http://doi.acm.org/10.1145/365559.365617

[3] C. A. R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (August 1978), 666-677. DOI=10.1145/359576.359585 http://doi.acm.org/10.1145/359576.359585

[4] C. A. R. Hoare. 2004. *Communicating Sequential Processes*, current edition published on-line at http://www.usingcsp.com/.

[5] G. Kahn and D. B. MacQueen. 1977. Coroutines and networks of parallel processes, Information Processing. In *Proceedings of IFIP Congress,* 77:993-998. http://hal.archives-ouvertes.fr/inria-00306565/.

[6] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms,* pp. 193-231. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[7] Dennis M. Ritchie. The Limbo Programming Language, In *Inferno Programmer's Manual, Volume Two*, pp. 91-100. Vita Nuova Holdings Ltd., York, 2000.

[8] Jeff Sickel, and Paul C. Nordine. 2010. Effective Resonant Frequency Tracking With Inferno, In *Proceedings of IWP9*, 2010: 42-52.

[9] Sean Dorward, Rob Pike, and Phil Winterbottom. 1997. Programming in Limbo. In *Proceedings of the 42nd IEEE International Computer Conference* (COMPCON '97). IEEE Computer Society, Washington, DC, USA, 245-.

# Appendix: Work In Progress

The WIP session included the presentation and discussion of the following works:

- *A Bluetooth Protocol Stack for Plan 9,* by Richard Miller
- *A Plan 9 C Toolchain for the Altera Nios2 Processor A Plan 9 C Toolchain for the Altera Nios2 Processor,* by Richard Miller
- *New file system proposals,* by Francisco Ballasteros, Sape Mullender, Latchesar Ionkov, and others.
- *Dfs – A WebDav filesystem client,* by Steve Simon
- *Wsys(4): hosted window system,* by Jesús Galán López

This appendix includes the submitted abstracts for the WIP session.

# A Bluetooth Protocol Stack for Plan 9

*Richard Miller*

miller@hamnavoe.com

*ABSTRACT*

The Bluetooth family of communication protocols is similar in function to TCP/IP, but different in character. Intended for low power, short distance point to point radio communication between pairs of devices, with links set up (at least initially) with human intervention, it has no multi hop routing, no message broadcast, no domain name or address resolution; it does have a trusted device model established by a one time "pairing" operation. Unlike TCP/IP, in which the higher protocol layers are typically implemented in software on top of a primitive hardware link layer of simple datagram operations between hardware addresses, Bluetooth chips act as "black boxes" which establish and maintain high level multiplexed connections between multiple devices, presenting a complex command and event interface to the host OS which requires a 1,420 page specification to describe.

In spite of these differences, the Plan 9 network abstraction is sufficiently general for a usefully large subset of Bluetooth to have been incorporated in a fairly straightforward way, simply by extending `/net` with a new protocol directory `/net/bt`, supported by the new `btfs` synthetic file server running as a user level process. With no changes whatsoever to the kernel or to C library routines such as `dial`, `announce`, and `listen`, it is possible, for example, for a Plan 9 machine to share its file system with another via Bluetooth using this pair of commands, on server and client respectively:

```
aux/listen1 bt!*!42 /bin/exportfs
import -A bt!001122334455!42 /n/blue
```

A small change to the connection server `cs`(8) allows the use of a more "friendly" device name in place of the hexadecimal Bluetooth device address.

Some of the Bluetooth "profiles" (standard services which can be provided over a Bluetooth channel) can add useful functionality to Plan 9. For example, the HID (human interface device) profile is a minor variation of the USB HID specification, so that changing a few lines of source code in the USB keyboard and mouse driver `usb/kb` has made it work with Bluetooth keyboards and mice. A simple command-line client for the OBEX (object exchange) protocol has been written, to fetch or send single single files between a Plan 9 machine and a Bluetooth-equipped phone or other device. A future project is to write a file server implementing the OBEX FTP protocol, which would allow a Plan 9 machine and Bluetooth phone to browse each other's file system.

# A Plan 9 C Toolchain for the Altera Nios2 Processor

*Richard Miller*

miller@hamnavoe.com

*ABSTRACT*

The Nios2 processor is a "soft CPU" which can be instantiated as part of a user-designed system-on-chip on Altera's Field Programmable Gate Arrays (FPGAs). It has a simple RISC architecture, with a range of implementations of varying size and complexity. The tiny Nios2/e fits in <700 Logic Elements (a typical FPGA will have tens or hundreds of thousands of LEs) by omitting luxuries like hardware multiply and divide instructions, memory caches, MMU, and user/kernel mode. The (relatively) high performance Nios2/f adds all these features with a 6 stage execution pipeline incorporating dynamic branch prediction and more, to occupy about 3000 LEs. Between them is the Nios2/s, a compromise between size and speed.

In a recent project I adapted the Inferno emulator for the Nios2, running on top of the POSIX compatible eCos operating system. This was useful as a platform for writing demo and test programs for a client's rapidly evolving FPGA-based hardware prototype. However the hosted implementation was unsatisfactory in some ways: the extra and largely redundant operating system layer used a lot of resources on a small chip; the `gcc` compiler used with eCos is unwieldy and its generated code is not particularly good. Therefore as a step towards native Inferno on the Nios2 (and perhaps eventually Plan 9 as well), it seems worthwhile to retarget the Plan 9 C compiler toolchain for this processor.

Using the existing MIPS toolchain as a starting point, to date I've produced a compiler, assembler and linker (`nc`, `na`, and `nl`) for the maximal Nios2/f, and added instruction disassembly routines to the `libmach` library. Future work includes software emulation of the missing integer multiply/divide instructions for the Nios2/e, and emulation of double precision floating point (the Altera cores support only single precision) and 64 bit integer operations for all processors.

# IX: A file protocol for NIX

**WIP**

*Francisco J. Ballsteros*

*ABSTRACT*

It is common to use Plan 9 file servers through slow network connections. Tools like CFS, OP, and others try to help there. But it would be desirable to be able to work from remote terminals suffering poor-latency links, keeping coherency with a main file server when feasible, yet being able to work disconnected when there is no other way. IX is a protocol designed to connect a caching file system client with support for disconnected operation to a remote file server. At present it is still work in progress. The implementation is included in the distribution for NIX.

**Problem statement**

We have been using the Octopus Protocol, OP [1], to connect to Plan 9 file servers over high-latency network links. But, while this protocol makes it feasible to operate on remote file servers with better performance than 9P, there are still problems:

- There is no support for conditional retrieval of file contents. Caching clients might require to retrieve a particular file only if its own cached version is out of date.
- There is no support for moving a file within the file hierarchy kept in the server without forcing the data to pass through the client.
- Coherent operation can be improved. There is a coherency window but the protocol does not make a distinction between device files and regular files (such as those provided by fossil).

The first issue is serious for clients that support disconnected operation, because they have, by definition, to be aggressive regarding data caching. The second issue is an optimization, but is important because the difference in performance may be significant, and it is not unfrequent to move files among different directories within the file server. The third issue is critical if the protocol must be able to operate on remote name spaces.

**IX design**

IX is built by leveraging what we learned from Op, the existing code-base for 9P, and ideas as discussed previously in the community [2]. The protocol is built upon a few design guidelines:

- The underlying transport is a reliable, ordered, connection—similar to a TCP stream—, with flow control for network congestion.
- An RPC in the protocol is a series of elementary transactions. Such transactions are similar to 9P requests.

IX is built upon the concept of connection channels. It is extremely cheap to build or dismantle a channel, so that it is reasonable to create one for each RPC. Channels are duplex, and inherit the reliability and ordering properties from the underlying transport. A single connection is multiplexed among multiple channels so that there is no starvation for sending or receiving through them.

An IX client would create a new RPC by just creating a channel. This does not require communication with the peer, and is a local operation. Then, one or more transaction requests would be sent through the new channel. The last one is flagged to indicate that the channel write direction can be closed when the

request has been sent. Only the client can allocate new channel identifiers, which are local to the connection being multiplexed.

An IX server receives requests through allocated channels and processes sequentially all requests for a given channel. It ceases to process them when one fails or when one is flagged to be the last one. Whatever happens first.

Note that channels are different from 9P tags. Like tags, they identify a particular outstanding RPC, so that multiple RPCs may be in transit at the same time. But, unlike tags, channels permit huge amounts of data to be sent (concurrently with requests for other channels) and each direction in the duplex stream can be closed independently.

Once a client has sent the desired transaction requests through a channel (or perhaps concurrently with them), a client receives through that channel individual replies for all transactions sent. An error reply to a transaction indicates that the RPC is finished, and the channel is closed.

To identify files, IX relies on *fids* and *qids*, similar to those in 9P. But, unlike in 9P, the server defines which values are to be used for new fids. The client has to keep its own data structures for files, which means that it has no advantage by selecting fid numbers. On the other hand, the server might exploit fid values to improve the data structure used to keep and look up fids.

Because a channel implies a context for individual transactions, it is feasible to simplify 9P transactions for use in IX to avoid unnecessary duplication through the wire. For example, once a fid has been established for an RPC, it is not necessary to repeat its value for each following transaction. The set of simplifications made is described in the next section.

To support aggressive caching, a conditional transaction has been added to the set of 9P requests known by IX. This transaction, *Tcond*, asks the server if a piece of metadata for a file is the same, greater than, less than, or different than the given value. The same relational operation can be performed for multiple elements of the *stat* information for a given file. If the condition holds, the server replies with an *Rcond* reply. Otherwise, the server replies with an error indication; Thus, terminating the RPC.

**IX Requests**

Using the syntax of *intro*(5) for 9P, this is the set of transactions known to IX:

```
Tversion msize[4] version[s]
Rversion msize[4] version[s]
Tauth afid[4] uname[s] aname[s]
Rauth aqid[13]
Rerror ename[s]
```

Similar to *version*, *auth*, and error messages in 9P.

```
Tattach afid[4] uname[s] aname[s]
Rattach fid[4] qid[13]
```

Similar to 9P's. But here the reply carries a value for the resulting *fid*. That fid is assumed as context for further transactions in the same RPC.

```
Tfid fid[4] cflags[1]
Rfid
```

Defines *fid* as the fid to use for following transactions in the RPC. Also, this request can set or reset two different flags for such fid: *OCEND* and *OCERR*. Both are used to automatically clunk the fid. The former upon reaching the end of file on read transactions, the latter upon errors in the RPC.

```
Tclone cflags[1]
Rclone newfid[4]
```

Similar to the *clone* request in the original 9P. Unlike in that, the client only specifies flags for the new fid (see the previous request) and the server decides on the value for the new fid. Note that, as in many other transactions, the fid to use (to clone in this case) must be defined by previous requests in the RPC.

```
Twalk wname[s]
Rwalk wqid[13]
```

Similar to the *walk* request in the original 9P. The implicit *fid* is walked to the given name.

```
Topen mode[1]
Ropen qid[13] iounit[4]
Tcreate name[s] perm[4] mode[1]
Rcreate qid[13] iounit[4]
```

Similar to 9P's counterparts, but using a implicit fid value. In particular, *create* would walk the implicit fid to the created file, and open it, when successful.

```
Tread nmsg[4] offset[8] count[4]
Rread count[4] data[count]      may be repeated
```

The *read* request exploits that requests are sent through channels. It permits specifying a maximum number of replies for the request. Each single reply may contain no more than *count* bytes. This grants the server rights to stream replies up to a given limit. Also, a side effect of the reply is that it may clunk the fid if it was *OCEND*.

```
Twrite offset[8] count[4] data[count]
Rwrite count[4]
```

Similar to 9P's *write*.

```
Tclunk
Rclunk
Tremove
Rremove
```

These requests do not need any fields (other than their types), because the fid is implicit.

```
Tstat
Rstat stat[n]
Twstat stat[n]
Rwstat
```

In these, an extra bit is used in the qid type, indicating if the file is (part of) a device. Such files should not be cached at all.

```
Tcond cond[1] stat[n]
Rcond
```

The *cond* request converts IX requests into a microlanguage capable of making decisions. Here, *cond* is a relational operator and *stat* supplies one or more non-null fields for file metadata. The server is expected to apply the relational operator to each non-null field, and reply with *Rcond* only if the condition holds in all the cases. Otherwise, the server replies with an error indication: *false*.

```
Tmove tofid[4]
Rmove
```

Moves the file identified by the implicit fid to the directory identified by *tofid*, if permissions permit. Should the name have to change, a separate *Twstat* request must be issued.

**Examples**

This is an example retrieval for a file:

```
-ch0-> Tversion  msize 8192 version 'ix'
|ch0-> Tattach  afid -1 uname nemo aname main
<-ch0- Rversion  msize 8190 version 'ix'
<-ch0| Rattach  fid 0 qid (0000000000000054 692693930 d)

-ch0-> Tfid  fid 0 cflags 0
-ch0-> Tclone  cflags 3
-ch0-> Twalk wname acme.dump
-ch0-> Tstat
-ch0-> Topen mode 0
|ch0-> Tread nmsg -1 offset 0 count 8190
<-ch0- Rfid
<-ch0- Rclone  newfid 1
<-ch0- Rwalk wqid (0000000000efd4d2 100 )
<-ch0- Rstat   stat 'acme.dump' 'nemo' 'nemo' 'nemo' ...
<-ch0- Ropen qid (0000000000efd4d2 100 ) iounit 0
<-ch0- Rread count 8185 ...
<-ch0- Rread count 8185 ...
<-ch0- Rread count 8185 ...
<-ch0- Rread count 8185 ...
<-ch0- Rread count 946 ...
<-ch0| Rread count 0 ''
```

Note that sending a *Tcond* request before the *Tstat* request in this example might change the dialog so that data and metadata would only be retrieved if, for example, the qid path or version had changed with respect to the ones given in *Tcond*.

As an example of how the implementation for a client might look like, this is an excerpt from the client being used for testing:

```
ch = newch(cm);
xtfid(ch, rootfid, 0);              /* 0 == it's not the last request */
xtclone(ch, OCEND|OCERR, 0);
for(i = 0; i < nels; i++)
        xtwalk(ch, els[i], 0);
xtstat(ch, 0);
xtopen(ch, OREAD, 0);
xtread(ch, -1, 0ULL, msz, 1);       /* 1 == it's the last request */
/* fid automatically clunked on errors and eof */
fd = -1;
if(xrfid(ch) < 0){
        fprint(2, "%s: fid: %r0, a);
        goto Done;
}
if(xrclone(ch) < 0){
        fprint(2, "%s: clone: %r0, a);
        goto Done;
}
for(i = 0; i < nels; i++)
        if(xrwalk(ch, nil) < 0){
                fprint(2, "%s: walk[%s]: %r0, a, els[i]);
                goto Done;
        }
```

```
        if(xrstat(ch, &d, buf) < 0){
                fprint(2, "%s: stat: %r0, a);
                goto Done;
        }
        if(xropen(ch) < 0){
                fprint(2, "%s: open: %r0, a);
                goto Done;
        }
        offset = 0ULL;
        do{
                m = xrread(ch);
                if(m == nil){
                        fprint(2, "%s: read: %r0, a);
                        goto Done;
                }
                nr = wdata(fd, m->io->rp, nr, offset);
                offset += nr;
                freemsg(m);
        }while(nr > 0);
Done:
        /* the channel is deallocated by now (a last request was sent and
         * a last reply was received
         */
```

**Implementation status**

A server that exports its own namespace has been implemented for NIX. A client for testing that performs operations on a single file is also implemented. The actual caching client with support for disconnected operation is still under construction. See the nix distribution for access to the source code.

**References**


1. F. J. Ballesteros, G. Guardiola, E. Soriano and S. Lalis, Op: Styx batching for High Latency Links, *IWP9*, 2007.
2. J. Floren, R. Minnich and A. Mirtchovski, HTTP-like Streams for 9P, *IWP9*, 2010.

# wdfs – A WebDav filesystem client

*Steve Simon*

*ABSTRACT*

The design and implementation of a WebDav filesystem client for plan9 is described together with current and possible future changes to improve its performance.

## 1. Introduction

There was a flurry of excitement about WebDav [1] in the late 1990s with the hope that it would become a standard for remote web authoring, and, with the addition of DeltaV[2] , version control. It has since been widely implemented it has seen use as a more general remote file access protcol in some areas; A WebDav server is even available on plan9 using the Pegasus HTTP server.

WebDav uses an extension of the HTTP protocol to send small XML snippets sepcifying the transaction(s) required, thus any plan9 WebDav impmentation will need an XML [3] parser.

### 1.1. XML parser

A DOM model XML parser was written for another project and adapted to the needs of parsing a network stream. DOM model parsers read an XML file into linked data structures in memory and need only one pass over their input.

The parser is complete and stable. The XML parser accepts standard entity references, PCDATA, attributes and values, and preserves the element hierarchy. Comments are elided and there is basic support for XML namespaces. It does not currently support CDATA or attempt validation against a schema or DTD.

The DOM model allows simple traversal of the hierarchy but puts large demands on the malloc library for its data structures.

Experience has shown that if attribute names and values are held in a reference counted tree and string heap respectively this problem can be minimised. This is not done in *wdfs* as the xml snippets are never big enough to warant the optimisation.

## 2. Webfs extensions

*Webfs* , the plan9 HTTP client application, had to be modified in several areas to support WebDav:

- Support the new WebDav message types
- Expose more HTTP header information
- Improve webfs's authentication coverage (digest added so far)
- Support chunked encoding, used by many WebDav servers

Every attempt was made to do this in a sympathetic manner but it is still not clear whether the design of *webfs* as a seperate application rather than a library is the correct way to proceed.

## 3. 9P file server

A fairly traditional multithreaded file server was written based on the lib9p library in the plan9 distribution, this part of the system presented little difficulty.

## 4. WebDav protocol variations

As *wdfs* was tested against more servers it soon became apparent that the WebDav protocol is not as detailed as one might like. Simple attributes may be supported differently on each server - for example a readonly flag on a file, and others only appear when the server is in a specific mode (MicroSoft™ SharePoint). As a result *wdfs* always provides a reasonable minimum feature set – reporting file ownership, readonly flags and, last access times – adding other metadata where available.

The most disappointing variation was the inability of some servers to support GET or PUT methods with a range specification. The the entire file must be transfered as a complete file read-ahead or write-behind. Though this improves overall throughput it increases latency making simple edits on large files painful. Furthermore there does not seem to be a way to detect reliably whether this feature is supported or not.

As we pay such a penalty missing this feature on some servers a command line option is available to enable partial writes or reads (using a range specifier) where they are known to be available.

## 5. Performance enhancement

The performance of *webfs* can appear fair for local file servers or dreadful for remote or slow ones, there are several reasons for this.

RTT
: Large round-trip times slow remote filesystem response. One obvious solution is to merge reqests where possible, somthing the protocol allows. A simple implementation would be to send a 9p walk request as a single transaction - implementing the walk() interface to lib9p rather than walk1().

Read-ahead
: Read-ahead could be used to asynchronously transfer more of a files contents when only a small amount has been requested in the traditional manner. This is already performed to the extent of rounding up transfers to 8Kbytes and serving reads from a local buffer. This technique could be extended to requesting a directory scan of the terminal directory of a walk, in the hope that it might be needed.

gzip content-encoding
: Currently compressed encoding schemes are not supported by *webfs* , these could improve response when faced with low bandwith connections, though this is less of a problem these days and so it was not considered a priority.

Persistent HTTP connections
: Currently webfs does not support persistent TCP sessions, thus there is the significant TCP setup and teardown time cost to each transaction.

So far only an experimental volatile metadata cache has been written. This stores only the results of walks and directory scans. This simple step provided a significant improvment to the perceived performance of *webfs* . The cache uses a published heuristic which attempts to reduce the chances of entries becomming stale.[4] .

## 6. Applications

*Wdfs* has found some unexpected uses in daily life. It becomes a readonly SVN client for plan9 - a WebDav server is part of every SVN installation and experience has shown it is generally enabled.

Venti score storage - several cloud storage providers,[5,6] offer WebDav to access their storage, and even allow users free access to a few Giga bytes. The author uses one of these to keep his venti scores.

Two other mildly amusing tools have also spun off from this work: an XML re-indenter, and an xml to flat file converter; similar to the linux application xml2(1).

## 7. References


1. *TTP Extensions for Web Distributed Authoring and Versioning (WebDAV),* IETF (Jun 2007). RFC4918 (draft)
2. *Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning),* IETF (Mar 2002). RFC3253 (draft)
3. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler François Yergeau, *Extensible Markup Language (XML) 1.0,* W3C (Nov 2008).
4. Vincent Cate., ''Alex - a Global Filesystem.,'' *In Proceedings of the 1992 USENIX File System Workshop*, pp. 1-12 (May 1992).
5. http://box.net.
6. http://www.mydrive.com.

# Wsys(4): hosted window system

*Jesús Galán López*
yiyu.jgl@gmail.com

*ABSTRACT*

Wsys(4) is a program which serves a 9P file system similar to the union of rio(4) and draw(3). Its interface is similar to rio(4), but wsys windows are created in a host system, ie. they are X11 windows in Unix (the only existing implementation for the moment). Using wsys, the window manager of the host system can be used to manipulate windows running applications on the hosted system, instead of simply emulating a screen in a single window. Wsys is built on top of some Inferno libraries and takes some lessons (and many lines of code) from the X11 version of p9p's devdraw(1).

## 1. Introduction: hosted drawing devices

9vx, drawterm and hosted Inferno all share the need of a *drawing device* in the host system. This device needs to communicate with the host windowing system (X11, Windows, Cocoa) and serve a 9P file system. Mouse and cons devices (pointer and keyboard in the case of Inferno) need to read events from the host system and, again, serve some files. A similar function is performed in Plan 9 by rio(4), which serves files equivalent to those of the native devices to every window (except draw(3), which multiplexes itself).

In a hosted system (9vx, drawterm, emu), mouse, keyboard and drawing devices are implemented as part of the kernel. Slightly different versions of the same code are included in each of these programs. This code includes some portable libraries and also a system dependent part. Bugs propagate at different rate than bugfixes and it is difficult to keep the all the versions in sync. Additionaly, in the case of X11, multithreaded applications can be problematic. As a result, it is more convenient to run the X dependent code in a different process. This solution is put in practice in p9p, which uses devdraw(1) to interact with the host window system.

A similar approach is taken by wsys(4) although, contrary to devdraw(1), wsys does not use a custom protocol. Instead, it serves a 9P file system similar to the one found in rio(4). Figure 1 shows how wsys(4) runs together with a hosted system taking the role of some kernel devices.

## 2. Wsys

When run, wsys posts a handler to a 9P server (by default in $NAMESPACE, as p9p and libixp do), or optionally listens from a port. Mounting it is analogous to mounting the *wsys* service provided by rio(4), with the particularity that windows are not created in an application window or a terminal screen, but in the host window system.

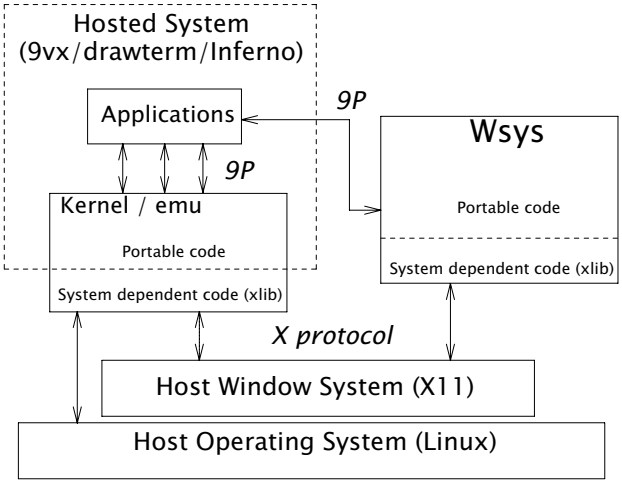



Figure 1. In a hosted system (drawterm, 9vx or Inferno) kernel devices contain system dependent code which communicates with the host window system (Xlib code in the case of the X11 version). `Wsys(4)` offers a similar file system from a different process which lets hosted applications to use windows in the host system through a 9P connection. As a result, the code dependent on the host window system can be taken out of the kernel

Most of `draw(3)`and `rio(4)` applies to `wsys`. When mounted, a new window is created and an entry is added to the `wsys/` directory common to all the clients. Files in `draw/` or the device and ctl files (`cons`, `mouse`, `wctl`, `label`, `snarf`, ...) work as expected. However, there are a few differences.

`Wsys` runs on the host, so it cannot create or finish processes on the hosted system. For example, when mounted from 9vx, `wsys` cannot create new processes inside 9vx, or kill any process when a window is closed. Creating new processes is actually not essential, because the method used by `window -m` is enough to use `wsys`. The `kill` file, discussed later, is used for process termination. As an additional utility, `wsys` is distributed with a `wctl` rc script which can be run from 9vx or drawterm to provide a *wctl* service, and be able to use `window` without the `-m` flag.

Another difference with `rio` is that `wsys`, when mounted with the attach specifier `/`, will not create a new window. In this case, a different file system is served, consisting on the `wsys/` directory with subdirectories for each open window, the common `draw/` directory (without `new` file), a `snarf` file, and a `kill` file. Reading the `kill` file blocks until a window is closed, and then returns the *pid* given in the attach specifier when that window was created. This file is read by the `wctl` script to kill processes of deleted windows. The `snarf` file, which is usually associated with the mouse in Plan 9, is served from `wsys` without being associated to any window. This is so in case some program (for example, the plumber) is interested in accessing the clipboard without having to create a dummy window.

### 2.1. Implementation

`Wsys` is built on top of existing technologies. Very few code had to be written from scratch. It includes some bits from Inferno, some bits from p9p (most of them from devdraw), some from rio, some from 9vx, ... The result of this combination is actually a

quite simple program, which is modular and portable, and does the job for which it was created.

Wsys makes use of a few Inferno libraries: `lib9` provides the basics, while `libdraw`, `libmemdraw` and `libmemlayer` are responsible of drawing on memory regions provided by the system dependent code. These libraries are also part of `emu` and, therefore, have already been ported to a number of systems and are well tested.

Libninep is the library responsible of serving the 9P protocol. This library is in fact an extended version of `libstyx(10)`, included with Inferno. `Libninep` adds to `libstyx` out of order requests, binding at the file tree definition level, and more control to process requests without using helper functions.

The only version of `wsys` available for the moment runs on top of X11. The Xlib dependent code has been mostly taken from p9p's devdraw, but also from Inferno and 9vx. The non-portable code is clearly separated from the rest, in order to make porting to other systems easier.

The rest of `wsys` is some code from `rio` for managing windows and a bit of glue to keep all the parts together.

## 3. Status and further work

`Wsys` is still evolving, but it already is in a quite usable state. However, it will not show all its potential until there are versions for other systems. Once OS X and Windows are supported, `wsys` could replace a big amount of code in Inferno, drawterm and 9vx.

For the moment, the X11 version of `wsys` still have some problems: resizing can leave a window blank, the keyboard can fail when going back and forward of fullscreen mode, exiting programs do not always delete the window... These bugs are not frequent, but they can happen.

Nevertheless, `wsys` can be used with 9vx or drawterm with reasonable success. With the help of the `wctl` script the usage of `wsys` feels very natural from the point of view of both the Unix host and the hosted Plan 9 sytem.

Inferno applications do not usually speak `9P` directly with devices, and instead they use `wm(1)`. However, `wsys` can be used, for example, to launch several `wm` instances in different windows:

```
; mount -Ab '#U*'/tmp/ns.$user.$DISPLAY/wsys /dev
; wm/wm &
```

Other possibilities are to use Inferno as a `drawterm` replacement using `wsys` and `9cpu`, or to use `wsys` in an Unix system to run applications from a *hellaphone* in X11 windows, for example.

Plan9port also includes code to deal with the host window system, but it cannot directly use `wsys`, since it currently uses its own protocol instead of 9P. Several solutions are possible, but none of them has been tried yet.

## 4. Acknowledgements

This work is the result of a Google's Summer of Code (GSoC) project. Thanks to everybody inside Google and *plan9-gsoc* for making it possible. Special thanks to Charles Forsyth for mentoring the project. Thanks also to the original authors of all the code used in the development of wsys. They are the authors of wsys too (but bugs are probably mine).