Networking Squeak

Bijan Parsia Bolot Kerimbaev Lex Spoon

August 17, 2000

Unless otherwise noted, the authorial "I" belongs to the primary author of the section, to wit, Bolot for the section on Comanche, Lex for the Squeaky Clients, and Bijan everywhere else.

1 Introduction

There is a apparent split in the Squeak worldview between the intensely individualistic and the thoroughly social. Squeak itself aspires to be a complete personal computing environment (with the single user in both computational and intellectual control from top to bottom) *and* a tool for collaborative development, exploration, and experimentation. This conception is akin to the notion of a networked personal computer—neither a thin client dependent on the network and server, nor an isolated workstation, but a node among peers, server, client, and self-sufficient in turn, separable but connected. A Squeaker is not merely autonomous, but autokoenomous.[1]

To support Squeaky autokeonony, Squeak has an extensive and varied set of networking facilities, applications, and frameworks, and a correspondingly extensive and varied community.

[1] "... 'autokeonony' which I take from the greek 'auto' ("self") and 'koinonia' ("community, or any group whose members have something in common"). What I mean by 'autokeonony' is "the self in community." pp., 145 Sarah Lucia Hoagland, *Lesbian Ethics*

1.1 Why use Squeak for networking?

You're looking around for a web server. Or maybe a new email client. Or perhaps you want to write a web crawler. Why use Squeak? After all, Squeak networking apps tend to be lacking in maturity—no surprise in so young a system. For example, Scamper, the web browser bundled with Squeak, is neither compliant with the latest specs, nor particularly polished, nor dramatically quick (though it is quite snappy for many purposes).[2] If what you wanted was the best or neatest or most powerful web browser as such, Scamper wouldn't fit the bill.

But Scamper (like every other Squeak networking app) derives some compelling advantages *simply* from being written in Squeak:

First, they share Squeak's hyperportability. Squeak images run (nearly) identically wherever there is a Squeak VM—no converting, recompiling, or tweaking needed. And since the Squeak's VM is very portable,[3] it has, in fact, been ported nearly everywhere. For general networking purposes, all that must be ported above the raw VM are the socket primitives.[4] Conceivably (and it has been proposed on occasion) this dependence on the host platform's networking capabilities could be eliminated by writing a TCP/IP stack in Squeak proper. (Anyone need a M.A. thesis?)

Second, they share in Squeak's malleability. Given the power and flexibility of Squeak's language and integrated development tools, it's very easy to program clients or servers, and even easier to tweak or extend existing ones. A few data points: The Pluggable Web Server, and the various Swikis built on it, were put together by Mark Guzdial when he was still new to Squeak, while Scamper and the IRC client were written by Lex Spoon during a Summer Internship at Squeak Central. As a Squeak, Smalltalk, and programming novice, I was, nevertheless, able to usefully modify all of these with very little effort.

Combined with its portability, Squeak's malleability makes it almost uniquely ideal for making custom, compact clients.

Finally, they share in Squeak's seamless systematicity. Every application—web browser, web server, mail client, what have you—is simply a collec-

[2] In the course of writing many things have changed in the Squeak networking world. For example, another web browser (a relic of Squeak project) has come to light, if not to general distribution.

[3] reference to Ian's chapter?

[4] Which, at the time of this writing, are themselves under some flux.

tion of classes which can be combined and reused in myriad and sometimes stunning ways. [5] It's easy to learn quite a bit about http, sockets, parsing, POP, and IRC just from playing around in Squeak. It's a great place to explore, experiment, and investigate.

In Squeak, you can be a programming dabbler and still have control over your tools and environment. Right now, you give up a certain level of maturity and functionality that you find in current commercial programs and most longer standing Free / Open Source software. But using Squeak now is like having been in on the ground floor of Linux—it's usable, it's only getting better, and it's getting better fast.

1.2 Why use networking in Squeak?

You're a Smalltalk veteran. You love hacking the compiler and the VM. You mess with meta-objects and pounce on primitives. Why muck with this mundane networking stuff?

Squeak aims to be a comfy computing home. We should be able to do just about anything we'd care to do, and, of course, one thing most of us want to do is play on the Internet. So, we should be able to browse the web, send and read email, chat, publish a website, and so on all from within Squeak, and it should range from pleasant to delightful to do so. Alas, while often pleasant and sometimes delightful, Squeak is not quite paradise. But it certainly has the potential, and that's a good reason to use Squeak for networking: to help realize that potential. So, there's the public service motive, and, in Squeaky parlance, there's a lot of "blue plane"[6] work to be done.

Squeak also aims to be a cutting edge research environment. Though some might say that "multimedia" is Squeak's focus, that's too narrow a description. Multimedia is important to Squeak, at least in part, because the various media are crucial to *communication* and, more generally, to *interaction* (not just "with the machine", but with ideas, other people, tools, and so on). Squeak is a good place to do serious and seriously fun exploration of various forms of collaboration, both with standard and with novel tools.

[5] Lovers of Cyberdog, Apple Computer's defunct component-based Internet suite can find much to like about Squeak.

[6] Is this in BTF?

At the time of this writing, most commercial projects using Squeak are focused on Squeaking the web. In general, there are a lot of projects for which using Squeak to build a website (for example) is an easy sell. Web apps are high profile and (with Squeak) a lot of fun.

There are more folks who know something about HTML, HTTP, FTP, and so on than who know something about Squeak. I've found both when learning and a teaching Squeak that having that solid ground to work from is very helpful. More than just providing some self-confidence and orientation, there's typically a lot of immediate gratification in using Squeak for networking tasks. Also, people typically have a lot of little network jobs that they'd love to have automated, which gives an immediate, pragmatic focus to the demonstration. It's quite pleasant to write little classes or even just a bit of workspace code to do the job that your disciple might have otherwise written a Python or Perl[7] script for—it gives your Python or Perl wielding friends a very nice taste of Squeak.

Historically, Squeak networking has had a delightful community of novice programmers, new Squeakers, gurus, and plain old end users. The people involved are a joy to be around. They also are an exhaustingly productive bunch. If "Net time" is quick, and "Squeak time" is quicker, then it should be no surprise that "Squeak net time" is blindingly fast.

Here are five ways for the native Squeaker to get acquainted with networking in Squeak:

1. use the many networking related classes available for various ad hoc, everyday tasks;

2. switch over one's Internetworking to the collection of end user network apps, tweaking and enhancing them along the way;

3. use networking type examples to introduce new folks to Squeak;

4. write new and interesting network based or network aware programs.

5. work on making the hodge podge of protocols, message formats, file types, name schemes, and so forth that make up our wired world as

[7] Python and Perl are both "scripting" languages often used for system administration, as well as for ad hoc, and not so ad hoc, networking tasks.

simple, natural, and transparent as possible to the programmer, user, and programmer/user alike.

There are plenty of people doing each of these things, so there's no lack of companionship, guidance, or audience.

2 Some simple SqueakWorking

So, you're ready to jump into the wonderful world of Squeak networking and you want to know where to begin? For your introductory pleasure, there is the Slick Demo and the Sober Overview. (Note: start with the Slick Demo.)

2.1 A Slick Demo

Most of the popular demos of Squeak involve doing crazy things with animation, 3D, sound, and so on, but I've found that for a lot of folks certain simple tricks have the biggest impact.[8]

All this demo needs is a Workspace, a basic handle on navigating the Squeak interface,[9], a net connection, and the "netchap.cs" change set filed into your image.

Start off by typing and inspecting the following in the Workspace:

```
'http://www.squeak.org/' asUrl.
```

[Display: HttpUrlInpsec.eps]

Cool! An URL is an object (hey, this is Smalltalk), and scanning the instance variables reveals a lot about the structure of URLs. When I first discovered this, I was grabbing URLs left and right, just to see what Squeak would make of them. (And it works for ftp:, file:, and mailto: URLs as well.)

[8] See Mark Guzdial's "Squeak demo for blowing students' minds" story at: http://minnow.cc.gatech.edu/squeak/52

[9] See chapter x. You need to be able open a workspace, enter text, select the text and be able to *do it* (alt/cmd-d, or the "do it" menu command), *inspect it* (alt/cmd-i, or the "inspect" menu command) or *explore* (alt/cmd-shift-i). A bit of familiarity with inspectors and explorers is also helpful.

When the charms of URL parsing wear thin, grab one of these URL inspectors, then enter and inspect following line in its "code pane":

```
self retrieveContents.
```

Whoa! Not only did Squeak go out and grab the web page at the end of the URL, but it doesn't just pop out the text—it represents the page as a **MIMEDocument**! What? Huh? Html files delivered over the web are **MIMEDocument**s? Hmm. I remember that now. In fact, inspecting

```
'http://www.squeak.org/SQ100x100.gif' asUrl retrieveContents.
```

will give me another **MIMEDocument**, only this time it's of type 'image/gif'. That sure beats guessing the type of the content from the filename extension.

[Display: MimeDocs.eps]

But, when we get down to it, the **MIMEDocument** version of the HTML is just a pretty wrapper around some text (the main type *is* text, after all). Can't Squeak do better than that?

Of course it can. Enter the following text in the code pane of an inspector of an HMTL **MIMEDocument**:

```
HtmlParser parse: (self content).
```

and Explore (cmd/alt-shift-i) it. If you have a GIF **MIMEDocument** handy, you can try doing the following, uglier code:

```
HTTPSocket showImage: (GIFReadWriter on: (RWBinaryOr-
TextStream with: self content) reset binary) nextImage named:
self url
```

(I find using the ObjectExplorer is more fun than inspectors for parse trees; I showed the ObjectExplorer to a friend and he shouted out, "Python should have this!!!")

After poking around the parse tree, and examining some of the parse nodes, "do":

```
self openAsMorph.[10]
```

[10] Bijan added method

[Display: HtmlParseTree.eps]

Now click on some blue text to open Scamper onto a new page. Yes, folks, those links are *live*.

[Display: RenderedHtml.eps]

This kind of demonstration shows a number of important points, even, I hope, to the most skeptical:

- First, and always, Squeak is dramatically cool—and not just for multimedia whiz-bangatry.
- Squeak is remarkably net-savvy, with a sensible understanding of URLs, MIME types, HTML, and so on.
- Squeak's net-savvy is accessible through a variety of powerful and interesting tools. There's something compelling about browsing through an HTML parse tree one moment and viewing the rendered HTML the next. The line between "programming tool" and "networking app" is rather thin, and this is an *advantage*, not a failing.
- This advantage isn't just productive, it's also didactic. What you know about networking guides you through the tools and the code. And one can learn about network "objects" (e.g., the structure of URLs) by using the Squeak development tools. Squeak provides a *uniform perspective* on your computing world. Opening an inspector on an URL is very much like opening an inspector on a **SmallInteger**. This systematicity, of course, is a driving design principle behind Squeak: *everything*, from the top to the bottom, from the interface to the virtual machine should be accessible, or perhaps *transparent* to the same set of tools and techniques.
- The final point is that some of those most skeptical types only yield to whiz-bang multimedia extravaganzas. Fortunately, Squeak can oblige them with a networkish twist.

[Display: FlyingScamper.eps]

2.2 A Sober Overview

2.2.1 The state of things

Through at least version 2.8, the core distribution of Squeak contains fifteen "Network-" categories comprising close to two hundred classes. Fortunately, a number of these classes are low-level, node style classes for HTML parsing, or support classes for applications so the actual number one needs to master is fairly low and, furthermore, the whole shebang is reasonably layered, allowing for task specific focused attention. Unfortunately, there are several useful network useful, indeed crucial, classes and methods in the image (e.g., there are several relevant Stream classes, many important String methods, and so on). When writing network stuff, it's often worth doing a little investigation of the image before plunging ahead with writing a slew of utility methods.

It's reasonable to divide, in a rough and ready way, the "Network-" categorized classes into 3 basic groups:

1. basic infrastructure, primarily the classes in the Network-Kernel and Network-Protocol classes;

2. middleware/support classes, such as those found in Network-Url, and such classes as **HtmlParser**, **HtmlFormatter**, and **MailMessage**; and

3. end user applications classes: **Scamper**, **Celeste**, **PWS**, etc. and their immediate helper classes (e.g., **CelesteComposition**).

The networking primitives are concentrated in two classes: **Socket** and **NetNameResolver**. **NetNameResolver** is a stand-alone utility class, which simply provides some useful address and hostname lookup methods on the class side (for example, NetNameResolver class>>addressForString:). If one is using a dial-up connection (or other situation where one's IP address may change frequently) when developing network applications, NetNameResolver>>localAddressString is very handy. Otherwise, one can pretty much leave **NetNameResolver** alone.

Even if one is mainly using higher level frameworks, some familiarity with **Socket** is useful for debugging and profiling. In particular, many fall into the following pitfall: Before one can successfully use any network service, the networking primitives must be initialized, typically with

Socket>>initializeNetwork. This needs to be done for every launch of the image; doing it more than once causes no harm, but it must be done at least once. Unfortunately, some platforms in some situations don't respond well to the initialization process if, for example, one does it before firing up the PPP connection.[11] To avoid these problems, Squeak doesn't #initializeNetwork at start up, which means that one must remember to do so before trying to do anything networkish. The end user apps all do this, but not everything else does. (So, be careful with Workspace experiments.)[12]

Socket provides a fairly standard suite of methods for doing normal socketish activities: pinging, connecting, waiting, reading and writing bytes, listening on ports, and so on. While it's perfectly reasonable to do socketish things directly with **Socket**, there are more Smalltalky facilities available, even for reading/writing bytes type stuff. Both the Comanche web/application server and the Flow internet/streaming framework[13] provide stream style access to sockets (and there are other **SocketStream** implementations floating about), and one should expect that, in the future, one of these will make it into the base distribution.

In the current image, subclasses of **Socket** (in particular, of its subclass **SimpleClientSocket**) are used to provide basic access to a set of Internet services, namely FTP, HTTP, SMTP, and POP.[14] **HTTPSocket** and **SMTPSocket** have some useful utility methods (mostly on the class side). For example, after opening an **SMTPSocket** on a mail server (using SMTPSocket class>>usingServer:), one can send arbitrary mail messages using #mailFrom:to:text:. The downside of this method is that you have to be aware of the textual format of mail message, and include *all* the headers (including "From:" and "To:", which might seem redundant). Similarly, while **HTTPSocket class** does have several convenience methods for fetching web pages (#httpShowPage:, #httpFileIn:, etc.), most of the helper methods require some nitty-gritty knowledge of HTTP header formats.

[11] VisualWorks initializes its network primitives at startup, and "Why does launching VisualWorks try to start up my PPP connection" is a very common question from Mac users of VisualWorks.

[12] It may be that this issue has been resolved by the time of your reading this. We can, at least, hope so.

[13] Both discussed below.

[14] The relevant classes are all in Network-Protocols.

Protocol	Middleware	End User / high level
FTP	ServerDirectory, ServerFile, RemoteFileStream, FtpUrl	FileList
HTTP (Client)	Some HTTPSocket class methods, HttpUrl	Scamper
POP	No really convenient wrappers.	Celeste
SMTP	No really convenient wrappers.	Celeste

Telnet and IRC are a bit different, which is no surprise having been written much later[15] with a somewhat different sensibility. Each is somewhat tied to Morphic, and not solely for the GUI, and thus presents a somewhat different programming interface (for example, the IO loops are implemented using Morphic's "stepping" feature).[16]

The URL classes (in "Network-Url") provide the beginnings of a nicer mode of access to various network services. Developed for Scamper, they do best for the retrieving of documents, but are easily extended to other tasks.

2.2.2 Flow into the Future

There is considerable interest in refactoring Squeak's networking system[17], and several projects working on different aspects. For example, at the time of this writing, there has been an extensive debate on the Squeak mailing list to determine the best semantics and implementation for the networking primitivesm including such issues as how to finesse the differences between different platforms, the advantages of pushing certain things down into the primitives, and how to ensure certain performance characteristics. At the other end of the spectrum, there has been considerable work done on Celeste, Comanche is gearing up to replace the Pluggable Web Server (PWS), a new Swiki framework has just reached beta 11, and so on.

There is also Flow, a comprehensive replacement for everything in between.[18] Flow is derived from a Squeak fork developed for a home networking research project. Thus, Flow is a mature framework, that was

[15] By Lex Spoon at the same summer internship where he wrote Scamper

[16] More on IRC below.

[17] To be fair, there is considerable interest in refactoring everything. *That's the XP (eXtreme Programming) way.*

[18] Flow includes its own set of primitives, but has been modified by John M. McIntosh to the use the existing primitives.

pounded on for several years as the substratum of a variety of networking applications.[19] Unlike the current system, Flow was developed by one person, Craig Latta, and so lacks the piecemeal feel that the current classes have. Flow is also quite complete, with support for sockets, files, MIDI, serial ports, and a full range of networking protocols. As of this writing, only the basic infrastructure has been ported and released, but that's enough to get a feel for Flow.[20]

The other significant fact about Flow is that it is the basis for a "Camp Smalltalk" project to provide a cross-Smalltalk Internet / networking framework. Essentially, Flow *is* that framework, so it is highly likely that it will show up in Squeak in some form.

Flow comprises three class hierarchies: a refactored **Stream** hierarchy, the **ExternalResource** hierarchy, and the **Correspondent** classes.[21] **ExternalResource**s are the low level classes which deal with the nitty gritty of encapsulating hardware details and OS facilities. "External" here should be thought of as roughly "external to the Squeak image", i.e., files, sockets, serial ports, USB, etc. These classes also wrap the requisite primitives—much as **Socket** and **NetNameResolver** do. Thus, unless one is implementing access to a new sort of input / output interface (e.g., IRDA), one should not have any need to modify or subclass, or even mess much with the **ExternalResource** classes. In general, one is better off reading and writing to them via the streaming protocols.

Indeed, streams are the heart and rationale of Flow, and, in some ways, the least obvious aspect. In general, streams provide a uniform interface for multiple and interruptible writing to, modifying, reading from, and enumerating collections of objects. For "internal streams" (e.g., on **Collection**s), streams abstract away from the idiosyncratic access details of the particular collection types. For example, **Array** and its subclasses (including, most notably, **String**) are of fixed size, whereas **OrderedCollection**

[19] Including a very spiffy web browser which also is being back-ported to the main Squeak distribution

[20] Currently released is "Flow 1" which covers sockets, but lacks the various network protocol support. Flow 2 does files, Flow 3 MIDI, and Flow 4 serial ports. All these were present and used in the Squeak port. Flow 5 is projected to handle FireWire / ILink connections. In what follows, unless otherwise specified, read "Flow 1" for "Flow".

[21] Flow 2 introduces a fourth hierarchy rooted in URL, which handles, as one might expect, URL style references to external resources. They are roughly compareable to the current **Url** classes.

is of variable size. Thus, without streams, to append a new object to the end of an **Array** requires copying the contents of the **Array** to a new **Array** whose size is greater by one, and then putting the new element in the last position of the new **Array**. Or, one could convert the **Array** to an **OrderedCollection**, #add: the element, and then covert back. Clearly, none of these strategies are really a good idea. They are error prone, performance poor, and just plain tedious. Fortunately, one may simply use a stream on the **Array** and start adding or inserting elements at will. Even better, one doesn't need to care whether the streamed over collection *is* an **Array**—the same stream methods will work for an **OrderedCollection**.[22]

Traditionally, the main external stream has been **FileStream**, which adds various external resource specific methods (#open, #close, #finalize, etc.), several file system information methods (#fullname, #directory, etc.), and "file mode" methods which influence how the underlying "collection of bytes" is to be treated (e.g., read only, write only, as characters with platform specific line endings, etc.). In Flow 2, **FileStream** is an **ExternalStream** on the **ExternalResource** subclass **File**. In Flow 1, this basic pattern is repeated[23] for all the various **ExternalResource**s, with one great simplification: both sockets and hardware ports are treated as kinds of **NetResource**, and have one concomitant stream class: **NetStream**. Essentially, **NetStream**s stream over external sequences of bytes which are interactively generated, i.e., via a request, wait, response sequence. In this way, **NetStream**s are analogous to **Random**—they are "generating" streams, rather than "iterating" streams.

Adding basic stream support for a new type of **NetResource**, whether new hardware or a new transport protocol, does not require modifying **NetStream** at all. One merely must implement in one's new **NetResource** some methods for reading, writing, and waiting for something to read or write.[24] **NetStream** does the rest. Of course, one may wish to add

[22]The performance advantage of streams is a classic Smalltalk tip. Flow incorporates several performancing tunings for external resources, including buffering. One nice thing about Flow's factoring of the Stream hierarchy is that these performance tunings typically come for free for new resources.

[23]Of course, there are more sophisticated types of streams one might want to have for reading and writing things other than simple sequences of bytes. For example, **NetMessage**s provide for streaming to structured text message formats, such as that defined by RFC822. But notice how in Flow, these are orthogonal to the resource streamed over.

[24]The actual four methods are, respectively, #next:into:startingAt:,

some convenience methods to **NetStream**, or a subclass, for managing the streamed over resource (much as **FileStream** does with #binary, #text, and so on), but for the actually streaming functionality, no more is needed.

With these two layers, Flow merely provides an elegant architecture for accessing and implementing rather low level features—even **NetStream** only deals with reading and writing bytes. It is in the third layer i.e., the **Correspondent**s hierarchy that supplies aid for implementing (and then using) the various Internet services. It supports sending and receiving email or news (POP and SMTP clients), browsing or serving web pages (HTTP client and server), interacting with chat servers (IRC) or a Telnet server, and makes it easy to add custom net protocols, printer drivers, connection and synchroniztion with palmtops, and so on. Flow separates protocol from transport. So, where squeak currently includes behavior for protocals like POP and HTTP in subclasses of **Socket**, Flow applications provide it subclasses of **Client** and **Server**. Unfortunately, Flow 1 includes none of the protocol specific classes, so it is a little hard to say what implementing and using them is like. However, the included abstract classes (**Correspondent**, **Client**, and **Server**) do provide a few hints, at least for implementation issues. **Correspondent** abstracts transport mechanism, stream creation, and connecting/disconnecting. **Client** adds support for making requests and sending commands to a server, whereas **Server** has some support for managing (multiple) client connections, server status, and responding to client requests.

Overall, Flow provides a remarkably well-organized approach to dealing both with new hardware and low-level system facilities, and with new application level protocols. It works hard to minimize repetition, tedium, and distracting details. It also embodies a coherent *model* of external, and thus network, services with cleanly separatedg concerns. While this particular model may not be to everyone's taste, clearly it is the mark against which alternative proposals will have to measure themselves.

#nextPut:from:startingAt:, #waitForReadabilityTimeoutAfter:, and #waitForWriteability.