2012 45th Hawaii International Conference on System Sciences

The Lively PartsBin—A Cloud-based Repository for Collaborative Development of Active Web Content

Jens Lincke[1] Robert Krahn[1] Dan Ingalls[2] Marko Röder[1] Robert Hirschfeld[1]

[1] Hasso-Plattner-Institut, Germany
{firstname.surname}@hpi.uni-potsdam.de

[2] SAP Research, CA, USA
dan.ingalls@sap.com

Abstract

The Lively Kernel is a browser-based environment for authoring active Web content. Being a self-supporting system, it is simple and compact, yet it supports rapid authoring with little need for programming expertise. Most importantly, the entire context of creation is now embedded in the cloud, a shared space in which to find, alter, create and share new content. The paper focuses specifically on a "PartsBin" metaphor for organizing Web content and on an incremental scripting metaphor for making active connections between components and for forging new functionality.

1 Introduction

The Lively Kernel [7] surprised and delighted many people by showing that it is possible to build a complete self-supporting computing kernel from nothing more than the language and graphics available in every browser. This held out a promise that "wherever there is the Internet there can be authoring". Experience showed, however, that a substantial gap remains between the programming environment afforded by kernels such as Lively, and what we usually mean by the word authoring. By authoring, we mean the creation of active web content in a manner that is similar to mashups or that involves, at most, experimentation with occasional snippets of JavaScript.

> "Kits and Concrete Manipulation: A kit is a set of primitive components, together with a framework for connecting the components to do new and interesting things. If objects built with the kit can in turn be used to augment the original set of components, then the range of application becomes very large, limited only by the capability of the primitive components and the manner of their interconnection." [8]

The focus of this paper is a higher-level platform built on top of the Lively Kernel, that provides exactly those facilities needed to create a wide variety of active web content:

A set of concrete components
Concrete here means visible and immediately manipulable and active.

A way to connect the components together
This too should be concrete – that is visible and immediately effective.

A way to save results as new components
Identify the parts included and their interface, serialize the whole, and store it back into the repository.

The remainder of the paper is structured as follows. Section 2 gives overview of the Lively Kernel. Section 3 discusses the repository of parts and Section 4 gives some implementation details. Section 5 shows an example of the PartsBin in use. Section 6 discusses related work and Section 7 concludes.

2 Lively Kernel

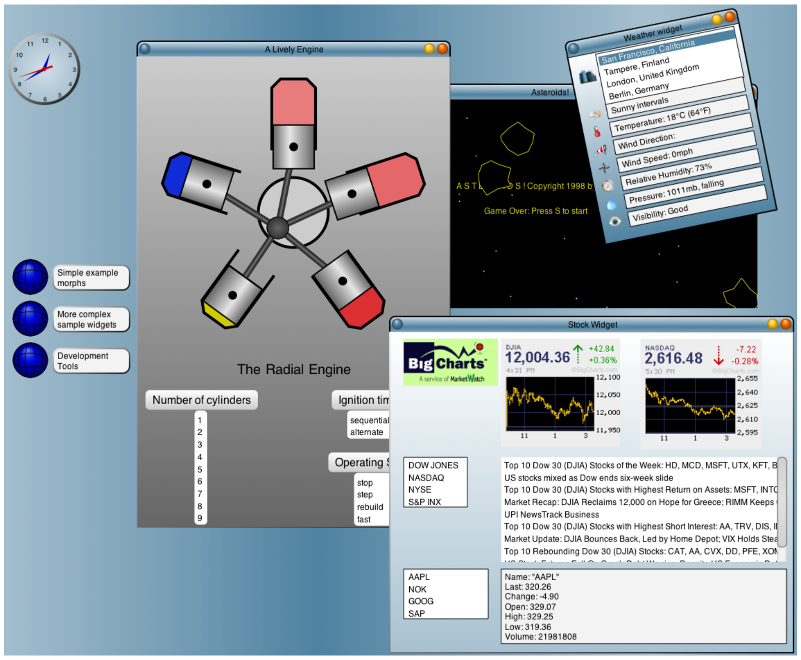
In this section we introduce our prior work, the development and runtime environment Lively Kernel.

2.1 Lively Kernel as a platform

The Lively Kernel begins with a Smalltalk-like class system [4], from which it defines a Morphic-style[1] [11, 12] graphics architecture, event handling,

[1] A Morphic architecture provides a family of primitive graphical objects that can be composed to build essentially any scene.

978-0-7695-4525-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2012.42



San Francisco, California
Tampere, Finland
London, United Kingdom
Berlin, Germany
Sunny intervals
Temperature: 18°C (64°F)
Wind Direction:
Wind Speed: 0mph
Relative Humidity: 73%
Pressure: 1011mb, falling
Visibility: Good

Top 10 Dow 30 (DJIA) Stocks of the Week: HD, MCD, MSFT, UTX, KFT, B
US stocks mixed as Dow ends six-week slide
Top 10 Dow 30 (DJIA) Stocks with Highest Return on Assets: MSFT, INTC
Market Recap: DJIA Reclaims 12,000 on Hope for Greece; RIMM Keeps
UPI NewsTrack Business
Top 10 Dow 30 (DJIA) Stocks with Highest Short Interest: AA, TRV, DIS, I
Market Update: DJIA Bounces Back, Led by Home Depot; VIX Holds Stea
Top 10 Rebounding Dow 30 (DJIA) Stocks: CAT, AA, CVX, DD, PFE, XOM

Name: "AAPL"
Last: 320.26
Change: -4.90
Open: 329.07
High: 329.25
Low: 319.36
Volume: 21981808

Figure 1. Screenshot of the Lively Kernel from 2008

and green thread scheduler. This in turn makes it possible to build active widgets such as text display and rich text editor, buttons and scroll bars, clipping frames and menus; in fact everything needed to build simple browsers, inspectors and debuggers.

The basic Lively Kernel platform proved to be successful. In less than 10,000 lines of code, it was able to perform all the functions of an IDE, in addition to providing interactive editing of its graphics components. Using SVG as a basis, it included its own window system for applications, and a linked-worlds model for major context switches. Network access was extended with WebDAV protocol, thus enabling direct access to an SVN repository in which the code was saved.

Beyond the fact that the Lively Kernel created "life" in any Internet browser, was the fact that it was only a Web page itself. It required no download or installation, but simply came to life when loaded by a browser. Able to edit its own code, it could save its applications, or even modified versions of itself as Web pages. It held out a promise that wherever there is the Internet there can be authoring.

Figure 1 shows a screenshot of the Lively Kernel as an application environment, here hosting four sample applications in its own window system.

Each node provides bounds, transformation, and ownership information, as well as functions that govern the response to input events, and redisplay of altered nodes in the scene.

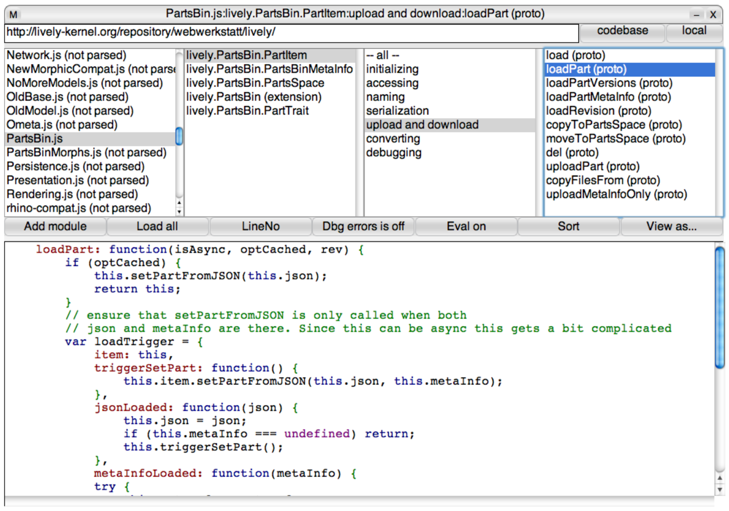
Figure 2 shows a comprehensive source code browser built in the Lively Kernel and used for serious programming. It offers direct access to an SVN repository, the ability to parse conventional JavaScript source code files into classes and methods, and to present them with syntax highlighting in the style of a Smalltalk System Browser [3]. In Figure 3 the browser is shown together with a model of a radial aircraft engine. When changes are made in the browser, they take immediate effect in the model, and are also updated in the repository.

An experienced programmer needs a kernel library, a language and IDE with which to build new components, and a facility for adding the results of his work back into the kernel repository. The original Lively Kernel served this purpose quite well.

2.2 Not the whole story

While appearing to be a breakthrough in openness, the original Lively Kernel was actually closed to many users. This is because unless one was a programmer, one could do little more than play with the graphics and applications supplied with the system. To build new content and extend the system required mastery of the relatively heavyweight, albeit lively, programming environment (see Figure 3)

Even in its original form, the Lively Kernel showed some real strengths as an authoring environment.



```
    loadPart: function(isAsync, optCached, rev) {
        if (optCached) {
            this.setPartFromJSON(this.json);
            return this;
        }
        // ensure that setPartFromJSON is only called when both
        // json and metaInfo are there. Since this can be async this gets a bit complicated
        var loadTrigger = {
            item: this,
            triggerSetPart: function() {
                this.item.setPartFromJSON(this.json, this.metaInfo);
            },
            jsonLoaded: function(json) {
                this.json = json;
                if (this.metaInfo === undefined) return;
                this.triggerSetPart();
            },
            metaInfoLoaded: function(metaInfo) {
            try {
```

Figure 2. The System Code Browser parses JavaScript source code and allows editing class and object definitions.

Worlds could be saved as projects with local code, thus allowing at least one level of convenient authoring. This capability was further amplified by a client-side Wiki [10] that allowed worlds to be stored into an SVN repository, thus making it easy to share projects, and to retrieve or revert to old versions whenever desired.

But saving projects and worlds was not enough, because it did not really provide the third requirement above, namely to store new creations back on an equal footing with other components in the library. Something was badly missing when it was so easy to make drag-and-drop creations in the Lively Kernel, yet it was so difficult to publish those creations where they could easily be accessed and built on by others.

3 Repository of Parts

As part of the drag-and-drop support for graphical creations, we had built palettes that allowed simple shapes to be easily located, grabbed and imported into a scene. Expanding on the SVN and Wiki support, we were also experimenting with server-side operations, inspired by the Node.js work [17]. At about the same time, we also succeeded in converting the entire system from a base of XML serialization to a cleaner and more general model built around JSON.

All of these factors converged to make it possible to serialize Lively components, complete with their graphics and code, store them on a server (as shown in Figure 6), and then retrieve them again in another Lively environment. Based on its early use, we called this component a "PartsBin", and it can be seen in use in Figure 4.

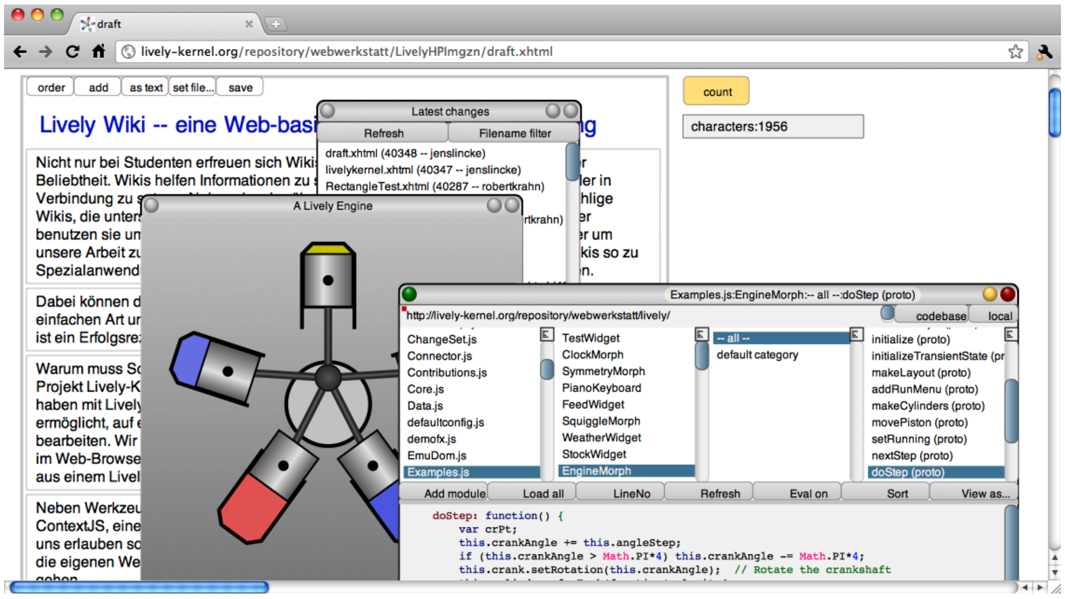
With a little bit of work, as shown in Figure 5, the PartsBin became an easily searchable, visually faithful collection of all sorts of lively components including

- simple shapes,
- UI widgets,
- windows, tabs, presentation structures,
- "vocabularies" such as slide show styles, musical components.

An interesting distinction between the PartsBin and other facilities for sharing user created content derives from the Lively Kernel's meta-circular design. Because the Lively Kernel platform itself is composed of malleable parts, new components in the PartsBin can change the nature of the environment and the applications that it can build. For instance a new tool in the parts bin might make it possible to design new gradient border fills that were not possible before, or it might provide new control over gridding not formerly offered in Lively worlds. This is the added dimension of extensibility provided by a system in which dynamic behavior is embraced rather than feared.

It was in this same way that we lifted our Self-sustaining System [7] development process up a level from programmer concepts like modules and classes to direct editing of end-user-accessible objects.

We began with two levels of editing content in lively:



Nicht nur bei Studenten erfreuen sich Wikis
Beliebtheit. Wikis helfen Informationen zu
Verbindung zu
Wikis, die unter
benutzen sie un
unsere Arbeit zu
Spezialanwend

Dabei können d
einfachen Art ur
ist ein Erfolgsre

Warum muss Sc
Projekt Lively-K
haben mit Lively
ermöglicht, auf e
bearbeiten. Wir
im Web-Browse
aus einem Livel

Neben Werkzeu
ContextJS, eine
uns erlauben so
die eigenen We

```
doStep: function() {
    var crPt;
    this.crankAngle += this.angleStep;
    if (this.crankAngle > Math.PI*4) this.crankAngle -= Math.PI*4;
    this.crank.setRotation(this.crankAngle);  // Rotate the crankshaft
```

Figure 3. A Lively Wiki page with a simulation of a radial engine and a System Code Browser viewing its underlying class.

- editing objects (properties, style, composition) [16]
- editing classes that indirectly changed the behavior of the object

We then moved from editing JavaScript source files in external editors to editing modules and classes from within Lively [10] as seen in Figure 3.

Meta-circularity is a double-edged sword, for editing each other's worlds and parts can break the system for all. What if someone breaks the ObjectEditor in the PartsBin? This can happen easily and then nobody can edit scripts of objects any more. Our approach does not limit these changes and parts can break, but working versions are still available in the repository. We follow the wiki principle that everyone can change anything but bad changes can be easily undone.

The stability of our environment is enhanced by working with concrete (fully instantiated) components. Since each part is copied when dragged out of the PartsBin, changing one part will not affect other parts, and breaking one object will not break other objects. This is different from a purely class-based system build by software engineers, who try to eliminate as much redundancy as possible. There, the shared behavior is defined in a class or superclass and modifying that behavior affects many objects at once. By fully copying a part instead of just instantiating a class we increase the redundancy in our system but make it more stable at the same time. Changing properties or behavior of an object affects only a specific object and does not have other unanticipated side effects.

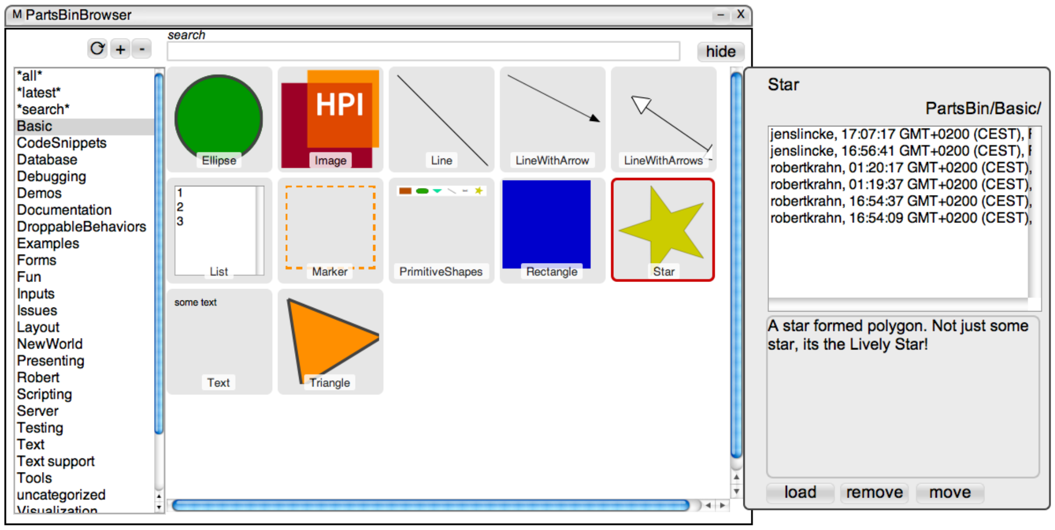
Sometimes users want changes to take effect in many places, for example a bug fix for one object should be propagated to other copies of that object, even those copies that exist in other worlds. Currently, this requires applying changes manually, for example, when extracting behaviors from a part into classes. The handling of sharing and inheritance is a deep design issue for a general authoring system such as ours. It is a goal of our continuing research to develop understandable controls over the apparent conflict between independence and dependence among components in an extensible authoring environment.

4 Implementation

Lively parts require the serialization of groups of objects with persistent state, behavior and module dependencies.

4.1 Persistent Object-specific Behavior

By allowing objects to have persistent behavior (as supported by the "addScript" capability) we could combine both material and procedural forms of authoring



jenslincke, 17:07:17 GMT+0200 (CEST),
jenslincke, 16:56:41 GMT+0200 (CEST),
robertkrahn, 01:20:17 GMT+0200 (CEST),
robertkrahn, 01:19:37 GMT+0200 (CEST),
robertkrahn, 16:54:37 GMT+0200 (CEST),
robertkrahn, 16:54:09 GMT+0200 (CEST),

A star formed polygon. Not just some star, its the Lively Star!

Figure 4. A PartsBin with the "Basic" category open. The pane on the right allows to view the edit history of the part, load old versions, move the part to a different category, or remove it.

content. Editing objects allows one to create content, form, and behavior. Self [19] explored this unification of state and behavior many years ago and our implementation language JavaScript builds on the same underlying concepts.

One difference of the unification of behavior and state in JavaScript and Self is that the behavior is transient in JavaScript since normal JavaScript closures cannot be fully serialized. We solved this by using our own form of closures when adding scripts to objects. With them we can inspect and serialize the instance-specific behavior of objects as well.

4.2 Modules and Parts

We have found it convenient to maintain a dual strategy for managing content in the Lively authoring system. Typically the user-built components are stored as fully-instantiated objects, whereas the system and primitive library components are stored in a module system that allows for dynamic loading of new code into the system as needed. Each module can pull in classes and modules that are needed to deserialize it. This enables us to package foreign resources and external libraries. We put thin objects into the PartsBin that require these libraries and make use of them. A good example of this is our integration of Protovis [1]. We do not need to load Protovis or even know about it before. The visualizations know which classes are needed to deserialize them and the classes in turn require the Protovis libraries.

While this dual strategy seems to be inherent in the different ecologies of kernel and user-built components, we continue to seek a solution that would more elegantly address the apparently different usage scenarios.

5 Creating applications with the Lively Kernel

Creating applications with the Lively Kernel involves combining and augmenting existing objects from the PartsBin. The resulting object has the same properties as any other part of the system and it can be published in the PartsBin as well. In this section we give a detailed example of the creation process of such a part.

5.1 CPU Visualization

In the following we show how to create a typical application using the PartsBin and existing parts. The logic of most applications can be broadly divided in three parts: Gather data from a (external) source, process and filter the data, output the result, for example in the user interface.

The application that will be created is a CPU visualization. It will access CPU workload data from the lively-kernel.org server, process it, and display the data using a Protovis [1] diagram.

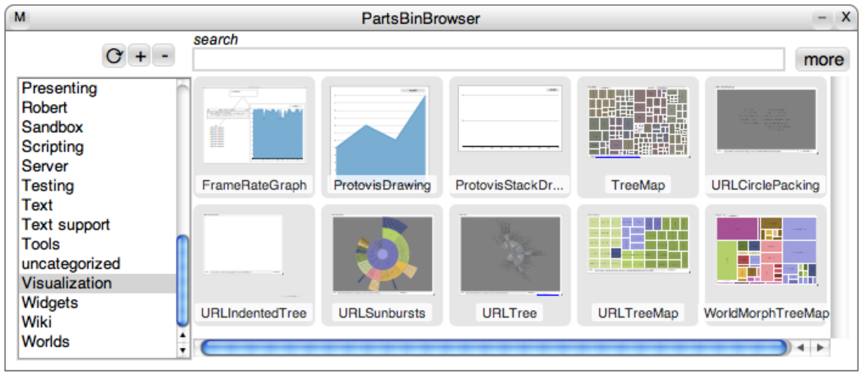
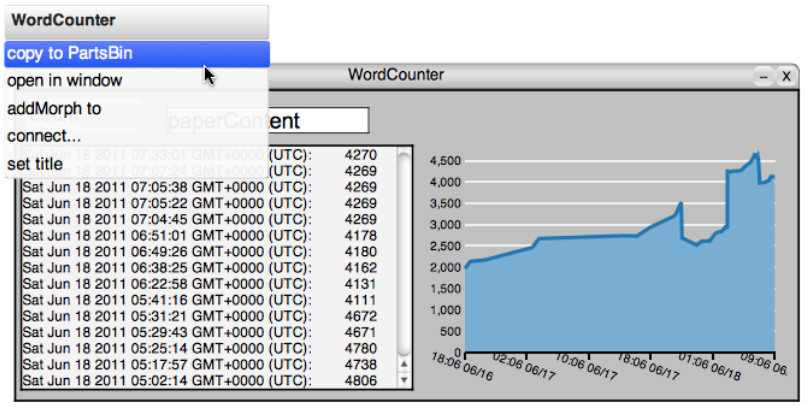


Figure 5. A PartsBin with the "Visualization" category open. These parts use Protovis and can be dragged out to be used or adapted if necessary.



```
Sat Jun 18 2011 07:33:01 GMT+0000 (UTC):    4270
Sat Jun 18 2011 07:07:24 GMT+0000 (UTC):    4269
Sat Jun 18 2011 07:05:38 GMT+0000 (UTC):    4269
Sat Jun 18 2011 07:05:22 GMT+0000 (UTC):    4269
Sat Jun 18 2011 07:04:45 GMT+0000 (UTC):    4269
Sat Jun 18 2011 06:51:01 GMT+0000 (UTC):    4178
Sat Jun 18 2011 06:49:26 GMT+0000 (UTC):    4180
Sat Jun 18 2011 06:38:25 GMT+0000 (UTC):    4162
Sat Jun 18 2011 06:22:58 GMT+0000 (UTC):    4131
Sat Jun 18 2011 05:41:16 GMT+0000 (UTC):    4111
Sat Jun 18 2011 05:31:21 GMT+0000 (UTC):    4672
Sat Jun 18 2011 05:29:43 GMT+0000 (UTC):    4671
Sat Jun 18 2011 05:25:14 GMT+0000 (UTC):    4780
Sat Jun 18 2011 05:17:57 GMT+0000 (UTC):    4738
Sat Jun 18 2011 05:02:14 GMT+0000 (UTC):    4806
```

Figure 6. A user-created WordCounter part to be uploaded to the server.

5.1.1 CPU Data Access

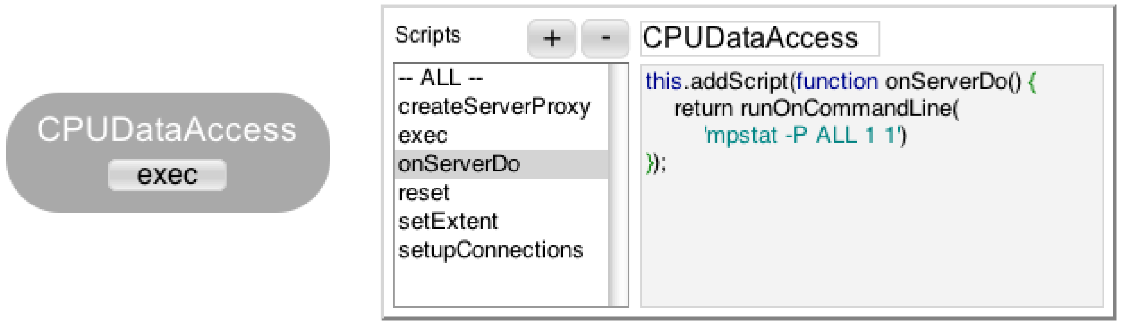
The first step when implementing the application is to access the data to be visualized. Lively Kernel provides a network abstraction that can be used to access any Web service via HTTP. However, for this specific application we need to run code on the server itself since there is no existing service that would deliver the required data.

We use a server [17] to run JavaScript code sent from a client in our environment. The interface to the Node.js server is encapsulated in a part. This part provides the general behavior that is required to send code and receive its result. Part developers can use this object by overwriting the *onServerDo* method to execute code in the server context. Figure 7 shows this part and how it was adapted to retrieve CPU workload data. Node.js allows one to run applications in the shell of the server operating system. Using the Linux's *mpstat* command line utility we access the CPU data. In the lower part of Figure 7 the output of that tool is shown. This output is also the result of the server request that is delivered to the client.

5.1.2 Processing and Visualizing the Data

For the visualization we employ another PartsBin object: A Protovis area chart. This object interfaces the Protovis JavaScript library and embeds it into the Lively environment. To feed the data from the server part into the visualization we use visual connectors that allow users to express a dataflow relationship between objects[2]. Part objects can provide *connection points*,

[2]These visual connections are established by the following simple gestures:
1. Choose "connect" from a menu of the source object, with



```
this.addScript(function onServerDo() {
    return runOnCommandLine(
        'mpstat -P ALL 1 1')
});
```

```
Linux 2.6.32-5-amd64 (lively)   06/19/11        _x86_64_        (2 CPU)
19:06:23     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
19:06:24     all    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.49
19:06:24       0    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.00
19:06:24       1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00

Average:     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest   %idle
Average:     all    0.51    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.49
Average:       0    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   99.00
Average:       1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
```

Figure 7. A part running code on a Node.js server to gather CPU data.

data sources, that, when changed, will distribute the data from their source to connected objects. The server part provides the source connection point *serverResult* and the Protovis visualization object the connection point *addDataAndRender*. The result when both points are connected is to add a data value to the visualization and re-render it whenever a server request from the CPU data accessor returns.

However, we need to refine the connection in order to adapt the server output to something the visualization expects and to select the data we are interested in (the CPU workload). In Figure 8 you can see the connected application and a converter that customizes the server-visualization connections. It gets a string similar to that from Figure 7 as input, parses it into a table, converts it into a number, and changes the idle value to the actual workload value by subtraction from 100. The converter can be added or altered by simply clicking on the visual server-visualization connector.

To enable the newly created application to run continuously we add a timer to it that then is visually connected to the server part. Whenever the timer ticks, its *tick* connection point is updated and it will trigger the *exec* connection point of the server part that will in turn issue a new server request.

For the final application, the parts needed only for construction (timer, server access) can be hidden. The remaining objects can be selected, named, and then published to the PartsBin as a new application.

outgoing property name from a submenu
2. Extend a rubber-band line to the target object
3. When the target has been chosen, choose the ingoing property name from the resulting connection menu

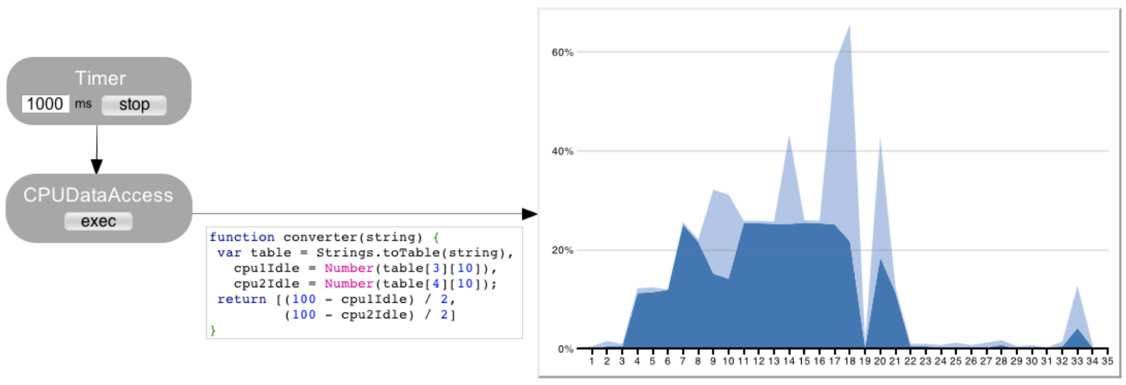
6 Related Work

Fabrik [8] is a visual programming environment that allowed one to program by visually connecting components dragged out of a PartsBin. The component composition could then be put back into the PartsBin for use elsewhere.

The Digitalk Parts Workbench [2] offered a similar facility for component layout and connection for the Digitalk Smalltalk environments. Its library was also later extended to wrap components in COBOL and Java.

Self [19] is an object-oriented programming environment that allows for programming objects directly. Objects in Self contain and access data and behavior in a uniform way. Objects in Self can be transported from one Self world to another [18], but Self does not come with shared repositories of objects.

Second Life [15] is a virtual 3D world where most content is generated by users. User can create and modify graphical objects which are made of graphical primitives and scripts. These objects can then be shared with other users by selling them or giving them away for free. Granting editing rights allows users to work together on the same objects. Since the creation



```
function converter(string) {
 var table = Strings.toTable(string),
   cpu1Idle = Number(table[3][10]),
   cpu2Idle = Number(table[4][10]);
 return [(100 - cpu1Idle) / 2,
         (100 - cpu2Idle) / 2]
}
```

Figure 8. The final CPU visualization and the parts for ticking and server access.

of active content is domain-specific the second life user interface and tools can not be extended from within the system as is possible with the Lively approach.

Squeak [6] is a Smalltalk-based programming environment that can be used as a personal multimedia authoring environment [5]. It has also a PartsBin, where objects can be dragged out. Squeak is a class-based system and programming objects directly is not the intended way to extend the system, except in the case of the EToys framework.

The SuperSwiki [14] allows users to share Squeak Etoy [9] projects over the internet. Projects are containers, similar to Lively worlds, for user-created content built from objects and scripts. Projects can be shared and modified, but can not be used as building blocks themselves. Our serialization approach is similar to the object streams used in Squeak. We made our data format robust to code changes since we use JSON as a format and not binary code that depends on a specific class layout, as Squeak projects do.

Many End-user programming environments have built-in repositories to allow their users to share content. A representative example is Scratch [13], a multimedia tile scripting programming environment for children that evolved from the Squeak Etoys work. In Scratch, sharing and remixing of projects plays an important part for being a "more social" programming environment. Children are encouraged to publish their projects from Scratch directly to the Scratch community website[3] where children can directly play the projects (via a Java plugin) and download them. As in Etoys, the unit of sharing is the project. Individual objects can only be shared indirectly through them, searching a PartsBin for objects and directly dragging them out as in our approach, seems to make the sharing and remixing a more integral part of the content creation process.

Yahoo! Pipes[4] is a Web-based end-user programming environment, that allows users to graphically wire together components to mashup Web-resources and produce RSS feeds. Those pipes can be shared and copied, modified, and reused by other users. Before a user can use or modify a pipe, the user has to copy the pipe to its own set of pipes. Pipes is domain-specific and therefore does not allow the creation of general active Web content.

The main difference between these environments and the Lively Kernel is that they do not allow one to create objects and tools that evolve the system itself. All of these environments distinguish between a tool and the material level: They put the editor as a tool on a different level as the script that is edited. This has the effect that there is no way to build a better editor out of the scripts in these worlds. It is clear that not all users want to have such a level of freedom and power, but building such freedom into a system allows us to evolve it in unanticipated ways.

7 Conclusion

It is gratifying when a solution to one problem turns out to nicely solve a number of other problems. We began with the simple need for a shape palette that was backed up on a server. But immediately it added the value of sharing. And when serialization was improved it became possible to store any component there, including complex tools and applications. Once those components could be accessed in the cloud, less needed to be included in the base system.

[3] `http://scratch.mit.edu/` (visited 2011-06-19)

[4] `http://pipes.yahoo.com/` (visited 2011-06-19)

Beyond merely solving more problems, the PartsBin repository has become a central element that simplifies the structure of the system and even suggests a new definition of the kernel. The Lively Kernel we once knew is now deconstructed out in the cloud, and growing beyond its former bounds. It is a shift in paradigm to where the "kernel" can now be reconstructed as simply that which is needed to support and work with a vast ecology of parts in the cloud.

What's next? We are at a similar place now with the ability to spawn active worlds in cloud resources, to visit headless worlds remotely, and to host live collaboration in our worlds. Hopefully we will soon be able to report some similarly practical results in the areas of collaboration and distribution.

References

[1] M. Bostock and J. Heer. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15:1121–1128, 2009.

[2] G. Bosworth, M. Teng, and J. Anderson. The Digitalk Parts Workbench. see `http://en.wikipedia.org/wiki/Visual_Smalltalk_Enterprise#PARTS_Workbench` as of 2011-09-15.

[3] A. Goldberg. *SMALLTALK-80: The Interactive Programming Environment.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[4] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

[5] M. Guzdial and K. Rose. *Squeak, open personal computing and multimedia.* Prentice Hall, 2001.

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices*, 32(10):318–326, 1997.

[7] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel A Self-Supporting System on a Web Page. In R. Hirschfeld and K. Rose, editors, *S3 2008*, LNCS 5146. Springer-Verlag Berlin Heidelberg, 2008.

[8] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, and K. Doyle. Fabrik: a visual programming environment. *SIGPLAN Not.*, 23(11):176–190, 1988.

[9] A. Kay. Squeak Etoys Authoring and Media, 2005. as of Aug 01, 2005, `http://www.squeakland.org/pdf/etoys_n_authoring.pdf`.

[10] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz. Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In *WikiSym '09*. ACM, 2009.

[11] J. Maloney. An introduction to morphic: The squeak user interface framework. *Squeak: Open-Personal Computing and Multimedia*, 2001.

[12] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28, New York, NY, USA, 1995. ACM.

[13] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for All. *Commun. ACM*, 52:60–67, November 2009.

[14] M. Ruger. SuperSwiki-Bringing Collaboration to the Class Room. In *Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings. First Conference on*, pages 18–21. IEEE, 2003.

[15] M. Rymaszewski, W. J. Au, M. Wallace, C. Winters, C. Ondrejka, B. Batstone-Cunningham, and P. Rosedale. *Second Life: The Official Guide.* SYBEX Inc., Alameda, CA, USA, 2006.

[16] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web Browser as an Application Platform: The Lively Kernel Experience. Technical Report SMLI TR-2008-175, Sun Microsystems, January 2008.

[17] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14:80–83, November 2010.

[18] D. Ungar. Annotating Objects for Transport to Other Worlds. *SIGPLAN Not.*, 30(10):73–87, 1995.

[19] D. Ungar and R. B. Smith. Self: The Power of Simplicity. *Lisp and symbolic computation*, 4(3):187–205, 1991.