Munin: A Peer-to-Peer Middleware for Ubiquitous Analytics and Visualization Spaces

Sriram Karthik Badam, *Student Member, IEEE*, Eli Fisher, and Niklas Elmqvist, *Senior Member, IEEE*

Abstract—We present Munin, a software framework for building ubiquitous analytics environments consisting of multiple input and output surfaces, such as tabletop displays, wall-mounted displays, and mobile devices. Munin utilizes a service-based model where each device provides one or more dynamically loaded services for input, display, or computation. Using a peer-to-peer model for communication, it leverages IP multicast to replicate the shared state among the peers. Input is handled through a shared event channel that lets input and output devices be fully decoupled. It also provides a data-driven scene graph to delegate rendering to peers, thus creating a robust, fault-tolerant, decentralized system. In this paper, we describe Munin's general design and architecture, provide several examples of how we are using the framework for ubiquitous analytics and visualization, and present a case study on building a Munin assembly for multidimensional visualization. We also present performance results and anecdotal user feedback for the framework that suggests that combining a service-oriented, data-driven model with middleware support for data sharing and event handling eases the design and execution of high performance distributed visualizations.

Index Terms—Ubiquitous analytics, high-resolution displays, multi-display environments, distributed visualization, framework.

1 Introduction

Visualization has long relied on computing devices equipped with mice, keyboards and monitors for virtually all applications [1]. However, this state of affairs is changing as problems and datasets grow in size, complexity and time sensitivity. Today's big data analytics problems often require more than a single mind or a single device to solve, and, as a result, parallelism for both users and computing resources is becoming necessary. In addition, computer hardware is evolving, and now encompasses devices such as large wall displays [2], tabletops displays [3], and tablets. This potential can be harnessed into building *ubiquitous analytics spaces* for sensemaking.

We present Munin, a peer-to-peer distributed middleware for building such multi-device ubiquitous visualization and analytics environments (Figure 1). These environments require managing distribution of data, computation, visual representations, and interaction between participating devices. For example, an input gesture made on one device should be communicated to a computational unit, and the updated visual representation should then be immediately rendered by the connected displays. To achieve this, the Munin framework transforms all networked devices in the same physical environment into Munin *peers* that communicate using a multicast mechanism on the peer-to-peer setup instead of depending on a dedicated server. This increases the robustness and fault-tolerance of the system, decreases its coupling, and improves its ease and convenience of use. The framework has a three-tier layered architecture:

- **Shared State Layer:** Shared and replicated *associative memory* that contains shared objects to which peers can subscribe and publish, as well as a shared *event channel* where peers can produce and consume real-time events for input and coordination.
- **Service Layer:** Mechanism for extending the framework's run-time capabilities through dynamically-loaded (potentially third-party) services for input, rendering, and computation that are implemented by specific devices given their individual capabilities.
- **Visualization Layer:** High-level components for ubiquitous visualization, including shared database used for replicating data across peers, and a data-driven scene graph with device-dependent implementations.

The Munin[1] framework is similar in scope and goals to many existing middleware solutions for ubiquitous and distributed user interfaces (DUIs), such as ZOIL [4], i-LAND [5], and Shared Substance [6]. However, Munin is tailored specifically for high-performance data visualization: (1) it has a shared table for managing and replicating datasets; (2) its distributed scene graph delegates rendering to visualization services that may choose visual representations depending on device capabilities; and (3) it eschews a traditional application-oriented model in favor of a data-driven model where cross-cutting instruments and renderers (implemented as services) operate on the data.

Building distributed software in general is a difficult venture, so another focus of the Munin project is to derive a programming model for ubiquitous analytics and visualization development that makes the process as painless as possible. We believe that Munin's service-oriented and data-driven architecture is key in achieving this goal, and facilitates building robust, fault-tolerant, and reusable software

• *All three authors are with the School of Electrical & Computer Engineering, Purdue University, West Lafayette, IN.*
E-mail: sbadam@purdue.edu, fisher55@purdue.edu, elm@purdue.edu

1. Munin is one of the Norse god Odin's twin ravens (Hugin is the other); serving as Odin's eyes and ears, they fly out into Midgard every morning and return at night with tidings about the world.

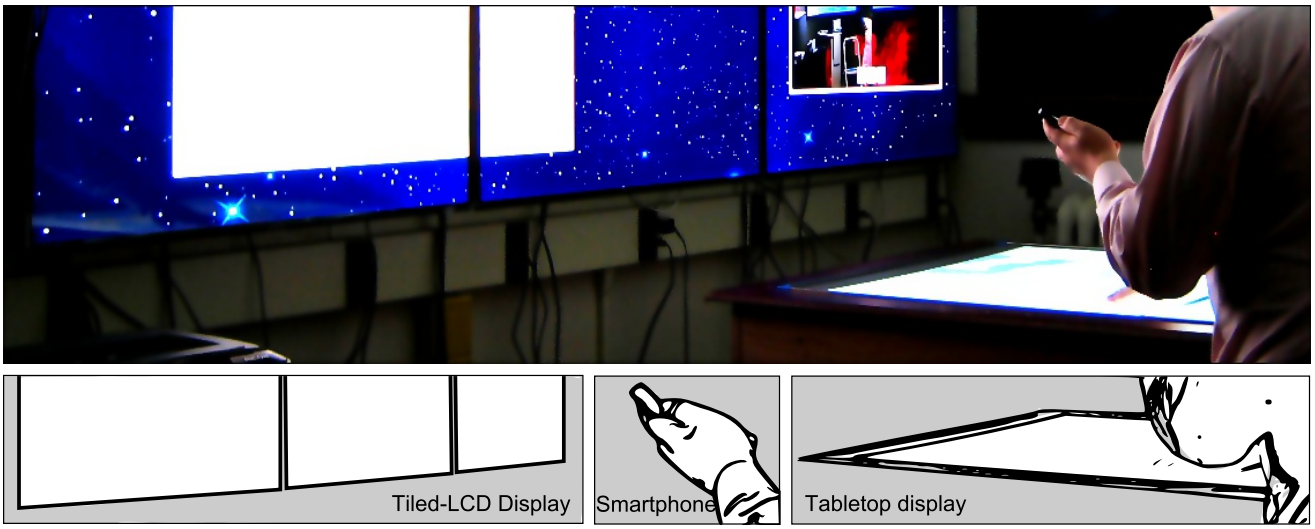


Fig. 1. A ubiquitous analytics space for a shared bulletin board built using the Munin toolkit and running on a tiled-LCD display wall, a digital tabletop display, and a smartphone. Touch input on the phone is used to create, move, and delete panels. The system consists of device-specific services that communicate over the network.

components that are loosely coupled yet combine to form a powerful system as a whole. Towards this end, we have implemented Munin in Java to facilitate easy deployment across different devices, and the networking support in the framework relies only on the JGroups P2P multicast library, which can be used even on mobile operating systems that support Java (now including iOS with the recently released Oracle ADF Mobile). This enables Munin to run on a range of devices such as smartphones, tablets, laptops, desktops, and all the way up to large tabletop displays as well as individual computers powering tiled-LCD display walls.

To validate Munin, we give three concrete examples of using the framework to build ubiquitous visualization software: a collaborative visual search environment combining mobile and tabletop devices, a geospatial visualization, and a distributed media player. We also give an in-depth example of how to build a Munin assembly using a multidimensional visualization for a large-scale display environment where multiple users can upload datasets, visualize them, and add scribbled annotations. In addition, we present the results of performance tests for the network and service layers of Munin. Finally, we present informal qualitative feedback from new end-user programmers adopting Munin for building new ubiquitous analytics (ublityics) spaces.

2 Background

Visual computing is broadly defined as the intersection of fields such as computer graphics, computer vision, visualization, and human-computer interaction, and is characterized by its emphasis on interactive visual representations. In this paper, we are particularly concerned with *visualization*, which is the graphical representation of data to aid cognition. As observed above, visual computing and visualization are changing. While historically confined to standard hardware such as a monitor, mouse, and keyboard [1], the scope is now widening for three main reasons:

(a) The problems that visual computing manages are growing in scale, complexity, and time-sensitivity;

(b) The need for collaboration is drastically increasing [7], as is the need to parallelize computation and rendering across multiple computers and devices; and

(c) Advances in computing technology has given rise to a new generation of *post-WIMP interfaces* [8] that significantly extend interaction capabilities past the traditional window, icon, menu, and pointer paradigm, as well as to *pervasive* and *ubiquitous computing* [9] where the computer is disappearing into the very fabric of our physical surroundings and everyday life.

Inspired by these challenges and novel computing paradigms, we here define the concept of *ubiquitous visual computing* as the integration of visualization and analytics capabilities into our physical environments and activities, also known as ubiquitous visualization and analytics [10]. A *ubiquitous visual computing space*, therefore, is a physical environment, either mobile (for nomadic use) or static (in a dedicated place), that supports ubiquitous visual computing through one or several networked devices that have been embedded into the environment. While this definition also encompasses standard personal computers, the spirit of these ubiquitous analytics spaces refer to post-WIMP devices of varying input and output modalities that are all networked to form a unified space for visual reasoning, analysis, and sensemaking. We formally characterize a ubiquitous visual computing space as follows:

C1 **Networked devices.** A ubiquitous visual computing space consists of one or several devices that are networked for cross-device analytics.

C2 **Collaboration support.** The canonical usage scenario for a ubiquitous visual computing space is for collaborative work involving multiple analysts [11].

C3 **Post-WIMP interaction.** Beyond traditional personal

computers into novel modalities such as touch, pen-based, gestural, and tangible interaction [8], [12].

C4 **Visualization and analytics support.** Visualization gives rise to unique challenges not commonly addressed in computer-supported cooperative work (CSCW) and ubiquitous computing (ubicomp) [11], including a data-driven model, multiple visual representations and views, and high-performance graphics.

To the best of our knowledge, this paper is the first to formally propose the idea of ubiquitous visualization spaces. However, our work builds on a plethora of existing research in the fields of computer graphics, visualization, human-computer interaction (HCI), ubicomp, and CSCW. Here we review this extensive literature; Table 1 summarizes relevant topics in context of our four criteria outlined above.

TABLE 1
Existing work relevant to ubiquitous visual computing.

Approach	C1	C2	C3	C4	Examples
parallel rendering	✓	–	–	–	[13], [14]
co-located groupware	✓	✓	✓	–	[6], [15], [16]
co-located visualization	–	✓	✓	✓	[11], [17], [18]
ubicomp toolkits	✓	✓	✓	–	[4], [19], [20]
visualization toolkits	–	–	–	✓	[21], [22]
novel vis. platforms	–	✓	✓	✓	[23], [24], [25]

2.1 Parallel and Distributed Rendering

Parallelizing rendering has long been a focus in the computer graphics community due to the fact that achieving photorealistic (or near-photorealistic) graphics quality used to be beyond the reach of individual computers (or, very expensive), while also being an "embarrassingly parallel" [26] problem that easily lends itself to partitioning subproblems to different renderers across the network.

Now that major technical advances in commodity 3D graphics cards have brought photorealistic graphics within the grasp of consumer the computer, the research focus of parallel and distributed rendering is instead to support very large and very high-resolution displays, often consisting of multiple tiled LCD monitors or projectors powered by computer clusters [27]. Current software that support this type of functionality include WireGL, Chromium [14], and Equalizer [13], but these distribute rendering at a very low abstraction level, typically within the graphics library. Even higher-level scene graph libraries such as SGI's Performer[2], VR Juggler [28], OpenSG[3], and OpenSceneGraph [4] provide little ubiquitous visualization support.

Beyond computer graphics, visualization was one of the early targets for research on parallel rendering, and much work has been done here, particularly for scientific visualization [29], [30]. Several of these projects go further by utilizing the networked rendering to distribute users as well [31]; we focus exclusively on co-located settings in this work, so such extensions are outside the scope of this paper. Nevertheless, the recent decrease in cost of high-resolution LCD displays has made creating large-scale display environments of such screens popular, resulting in several multi-display visualization frameworks being proposed [27]. Examples include Tiled++ [32], which eliminates distracting bezels by rendering content on them using a projector, and the Scalable Adaptive Graphics Environment (SAGE) [33], which is based on streaming dynamic graphics from different computers over the network to independent windows on a tiled display setup.

For virtual reality, Myriad [34] utilizes a peer-to-peer design to manage the individual scene-graphs in a cluster of displays. The scene graph nodes store the geometry, specify 3D transformations, textual maps, and OpenGL materials. It also provides a fine-grained sharing model through message filtering, reality mapping, and node locking. Shen et al. [35] investigate the peer-to-peer infrastructure for virtual world software to counter the architectural issues caused by client/server systems such as limited number of players in each server, single point of failure risk, and unbalanced computation resource. They survey the requirements of peer-to-peer virtual world design by discussing the various issues in those environments such as consistency, responsiveness, scalability, persistency, reliability, and security, along with existing work solving these issues through state management, overlay management, and content management.

2.2 Co-located Groupware

Groupware is the CSCW term for software designed for multiple users, and our focus in this paper is on groupware for *co-located* (same space) and *synchronous* (same time) settings. The branch of CSCW concerned with this kind of groupware is called multi-display environments (MDEs) [16], with several MDEs having been proposed through the years. Some early examples of MDEs were the Spatial Data Management System [15], Feiner's hybrid real/virtual interfaces [36], and the DigitalDesk tabletop [3].

Some MDEs inhabit entire rooms; examples include the CoLab meeting room [16], the i-LAND roomware system [5], and the Stanford Interactive Workspaces project [37]. The office of the future [38] project attempts to augment a room using geometrically-corrected projectors, structured light, and camera-based tracking.

One of the most closely related systems to Munin is the Shared Substance framework [6], which serves as the inspiration for the data-driven distribution model in our framework. However, Munin goes significantly beyond Substance's functionality with several mechanisms to support visual computing, including a shared data table, a dynamic service framework, and a distributed scene graph.

2.3 Co-located Collaborative Visualization

Collaborative visualization has many parallels to CSCW research, but also some important differences [11], including a data-driven and less document-oriented sensemaking

2. http://oss.sgi.com/projects/performer/
3. http://www.opensg.org/
4. http://www.openscenegraph.org/

process as well as a need for multiple views and multiple visual representations. Disregarding the distributed and asynchronous branch of collaborative visualization [31], there are many interesting problems and tremendous potential in adopting CSCW methods and techniques to visualization.

More specifically, while large displays have seen increasing use in collaborative visualization [11], it is only recently that novel devices such as tabletop displays were adopted for visualization; examples include tree comparison applications [17], multiple coordinated views [25], and mixed-presence visual analytics [18]. However, these systems are often one-off implementations, and not general toolkits.

2.4 Distributed and Ubiquitous User Interfaces

Distributed user interfaces (DUIs) distribute interface components across one or several of the dimensions input, output, platform, space, or time [39]. As we draw nearer to a vision of truly ubiquitous computing (ubicomp) [9], where information processing has been seamlessly and transparently integrated into everyday objects and activities, such interfaces are becoming increasingly important.

Several models for DUI and ubicomp interfaces exist, such as the CAMELEON-RT middleware [40], distributed versions of the model-view-controller (MVC) paradigm, and the recent VIGO model [19] for building distributed ubiquitous instrumental interaction applications. In terms of toolkits and frameworks, examples include BEACH [41], MediaBroker [42], and the Proximity toolkit [43]. Again, none of these are targeted at visual computing, and provide little support for high-performance rendering and computation. A recent toolkit for distributed user interfaces called ZOIL [4] may be an exception, but uses a client/server architecture, builds on Microsoft SDKs such as WPF and the MS Surface SDK, thereby restricting its portability, and has a fairly strict scene graph model based on semantic zooming. jBricks [44] is a Java toolkit for rapid development of applications on cluster-driven wall displays, but relies on a specific structured 2D graphics engine and does not provide the shared state, service-based architecture, and visualization constructs that Munin does.

Existing work by Bi and Balakrishnan [45] and Endert et al. [46] investigate the use of large displays and multi-screen environments in personal computing and standard office environments respectively. Bi and Balakrishanan [45] found that the usage patterns for large displays indicate that the users spend more time organizing the contents into focal and peripheral regions, thus establishing the need for automated/semi-automated layout management, legibility, and window management operations to save time. Endert et al. [46] discuss the various ways in which large displays can be used as extended memory. They also provide suggestions regarding the display configuration, keyboard placement, mouse placement, and user stance for different scenarios of large display use. Moreland [47] present various lessons learnt in using large-format displays and try to extend their use as a visualization space. We believe that these works provide the guidelines for developers building ubiquitous analytics applications through frameworks such as Munin that allow the management of display, input, and data in the ubiquitous analytics environments.

2.5 Visualization Toolkits and Novel Platforms

The visualization community has a long history of building toolkits and libraries to speed up development; examples include VTK, the InfoVis Toolkit [22], and D3 [21]. However, these are all targeted at the desktop platform with little support for ubiquitous visualization and novel devices.

In general, while these traditional platforms for visualization are now being challenged, there still exists very little prior art in this area. The most relevant work on post-WIMP interfaces [8] for visualization include data sonification [48], haptic and force-feedback visualization [24], and tangible interaction for visualization [49]. Bowman et al. [50] propose the notion of information-rich virtual environments (IRVEs) as the intersection of information visualization and virtual reality, but limit its scope to virtual 3D worlds, and not the real world itself. Perhaps the most active research topic here is the collaborative visualization on tabletop and wall-sized displays reviewed above. Finally, mobile visualization [51], [52], [53] translates visual representations to a mobile form factor to allow nomadic viewing and sensemaking of data. However, research here has mostly been focused on the limited size and computational power of the mobile device, and not on connecting multiple such devices for data analytics.

3 Design Space: Ubiquitous Analytics

Display space is quickly becoming a prime commodity as our datasets and applications become more complex, and this is doubly true for analysis and sensemaking settings [54]. Large displays have been shown to yield significant productivity increases even for standard office tasks [55], and as a result, multi-monitor setups are now common in the workplace. For more complex sensemaking processes, it may be as straightforward as more display area simply giving us more "space to think" [54]. However, when combining these large displays with multiple collaborators [11] and post-WIMP user interfaces that go beyond the traditional desktop [8], it is clear that we are entering an entirely new paradigm of *ubiquitous visual computing* where visual representations, analytics, and sensemaking are embedded in our physical surroundings and activities.

Such physical spaces are amenable to modeling using the new post-cognitivist theories that go beyond the traditional cognitive science view of the human brain as a pure information-processing entity where the surrounding world serves merely as a source for sensory input and a place for performing actions [56]. In a ubiquitous analytics space, it is easy to see how cognition is *distributed* [57] across multiple agents, display surfaces, and physical artifacts; how it is *extended* [58] by the social and material environment; and how it is *embodied* [59] in how we perceive and interact with these physical surroundings.

These criteria yield nine design requirements (R1-R9) for a software framework for ubiquitous visual computing.

3.1 Multiple and Networked Devices (C1)

A ubiquitous visualization space will consist of an ensemble of heterogeneous devices, ranging from personal computers to mobile devices. This yields several requirements:

R1 *Cross-platform*, to support heterogeneous devices.
R2 *Peer-to-peer*, to avoid servers and increase robustness.
R3 *Graphics-agnostic*, since devices have varying APIs.

3.2 Supporting Collaboration (C2)

Collaborative work is a canonical use-case for a ubiquitous visual computing space. This yields several requirements:

R4 *Multiple concurrent users* with concurrent interactions.
R5 *Co-located and synchronous* collaboration support.

3.3 Post-WIMP Interaction (C3)

Using multiple networked devices for both solo and collaborative work requires new interaction models beyond the mouse and keyboard. We derive additional requirements:

R6 *Multiple input sources*, such as touch, gestural, pen-based, tangible, and full-body interaction.
R7 *Multiple output sources*, such as wall-mounted, tabletop, mobile, volumetric, and tangible displays.

3.4 Supporting Visualization (C4)

Unlike current MDEs/DUIs, we target high-performance visualization with several unique requirements:

R8 *Visualization mechanisms and patterns* (e.g., [60]).
R9 *Multiple visual representations and views* [11].

3.5 Additional Design Considerations

The intended user for Munin is the application programmer (i.e., an "end-user" programmer) who is using the framework to build ubiquitous visualizations. However, distributed programming is inherently difficult, and so we also want to design the framework in such a way that it makes this process as painless as possible. Furthermore, the framework must be extensible to support multiple device modalities with different interaction types, to accommodate future new technology and to allow third parties to contribute new functionality, services, and visualizations. Munin's service layer resembles the tool layer of Vrui [61], a virtual reality toolkit that maps high-level semantic events such as navigation and menu selection to the input devices, thus making applications platform independent. Similar to IVTK [22], we adopt Alan Kay's philosophy for designing a framework so that simple things become simple to do, and so that complicated things become possible.

4 Munin: Overview

Munin is a software framework for building ubiquitous visual computing spaces. It is implemented in Java to facilitate running on different platforms (R1) and uses peer-to-peer (P2P) distribution for replicating state across connected *peers* (R2). Figure 2 depicts the Munin network architecture. The framework uses a distributed scene graph

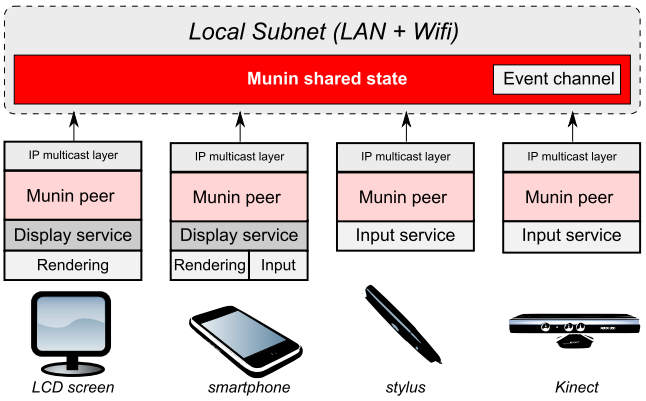


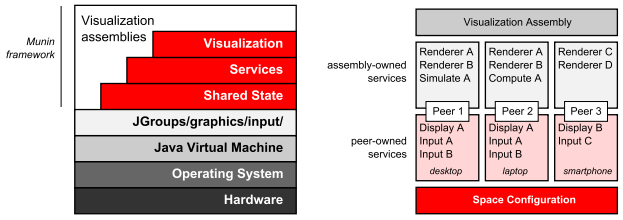
Fig. 2. Example Munin network architecture.

that is not tied to any graphics library (R3). Because of its networked architecture, Munin supports multiple concurrent users (R4) in a co-located and synchronous setting (R5).

Munin is a layered system (Figure 3(a)) with three primary layers: a Shared State layer (Section 5) for replicating data across peers, a Service layer (Section 6) for extending the framework's support for novel input (R6) and output (R7), and a Visualization layer (Section 7) for managing a distributed scene graph and high-level constructs for visualization (R8), including multiple views and visualizations (R9). The Munin design philosophy is to delegate the data management and rendering to the peers themselves. For this reason, the framework does **not** support any form of streamed graphics; all peers are provided with the data and scene graph to draw, and are expected to render their own visual representations. This minimizes coupling between devices and enables a robust, fault-tolerant, and truly decentralized architecture. Peers therefore run services optimized for the rendering, input, and computational capabilities of each particular device.

A collection of Munin peers in a specific physical environment (either a mobile or static setting) is called a *Munin space* and is a pure peer-to-peer system with no dedicated server. However, each Munin space has a global *space configuration* that maps out all peers, their capabilities, and their physical arrangements as well as the input and output surfaces they constitute. This configuration also maintains the peer-specific services necessary for a particular device to render, manage input, handle events, and perform computation. Since there is no central server for coordination, the space configuration is global and known by all peers so that each peer's own area of concern (in visual and data space) can be reliably determined.

Applications in Munin are called *assemblies* because they bring together a multitude of services running on different peers in the Munin space (Figure 3(b)). Some of these services are *peer-owned* in that they are launched in response to each peer's hardware configuration, whereas others are *assembly-owned* in that they contain the application and rendering logic of the assembly. This way, Munin eschews a traditional application model for a service-oriented one where data and services combine to form emergent behav-



(a) Layered system architecture. (b) Assembly architecture.

Fig. 3. Munin system and assembly architecture.

ior. This also promotes building generic and flexible services that can be reused in other assemblies. The example in Figure 3(b) portrays a simple Munin space consisting of three peers—a desktop, a laptop, and a smartphone—where the space configuration of each peer governs the peer-owned services (for example, both laptop and desktop use the same display service, whereas the smartphone uses a custom one), and the assembly has launched specific services for rendering, simulation, and computation. The Munin LaunchPad (Section 7) handles the details of launching services associated with an assembly.

5 Munin: Shared State Layer

The Shared State layer provides the basic network communication for replicating state and events across all peers.

Shared Associative State. Inspired by prior work on tuple spaces [62] and event heaps [63], Munin uses a shared *associative memory* that contains shared objects to which peers can subscribe. Subscribers will automatically be notified of changes to a shared object. This enables peers to create, modify, and update shared state that is replicated across all of the devices that constitute the Munin space.

In our implementation, the shared state consists of a single namespace addressed using UUIDs. Shared objects consist of dynamically typed properties, and can be created, read, modified, and deleted by any peer. Supported data types include any Java data type that can be serialized. In particular, properties can contain UUID references to other shared objects, allowing for complex data structures.

Shared objects are the main form of communication and synchronization for services in other layers of the framework. The fact that our shared objects are dynamically typed is both a strength and a weakness. Dynamic typing makes for more flexible data modeling and means that the developer does not have to define interfaces for all data exchanges. However, the disadvantage is that all data exchanged between services must be manually checked.

Shared Event Channel. Munin incorporates a shared *event channel* where peers can distribute real-time events. Exposing the event system in shared state allows for easily decoupling input and output subsystems, which is common in ubiquitous computing environments—consider, for example, a tiled display wall consisting of multiple computers responsible for output, but only a single computer managing gestural input from a Vicon motion-capture system.

In practice, shared events are similar in implementation to our shared objects with the exception that events are not persistent. Furthermore, because of the framework's peer-to-peer architecture, there is no deterministic event propagation order, and there is no way for a peer to consume an event (i.e., remove it from further propagation). Peers use the space configuration to independently determine whether a particular event falls within their area of interest.

Persistence. Since Munin lacks a central server, the shared state and event channel is replicated on all peers. This means that the space itself, including all state, disappears when the the last peer in a Munin space is shut down. However, the fact that the shared state is fully replicated across all peers means that it is straightforward to persistently store a snapshot of the shared state. All shared objects are already required to support serialization, so this is simply a matter of iterating through all shared objects and serializing them (including their UUIDs). The reverse process can then be performed by that peer when restoring persistent state on startup of the Munin space.

6 Munin: Service Layer

The Service layer provides a mechanism for dynamically-loaded *services* that extend the Munin framework in several different ways. In fact, without core services from the Service layer, a Munin peer is simply an empty service platform with no functionality beyond the shared state and event channel. While it is possible to develop ubiquitous visual computing spaces based only on this core, Munin provides a foundational set of services for input, output, computation, and event management.

Service Management. Munin services are implemented as Java plugins using JSPF (Java Simple Plugin Framework) that are dynamically loaded at run-time as JAR files from a local directory or over the network. This mechanism can also be used to download and upgrade service implementations. While JSPF does not support unloading plugins at run-time, it does allow for adding new plugins at any time, so that services can be hot-swapped while the system is running by simply creating a new version of a service. JSPF uses Java annotations to minimize the boilerplate code needed to write new plugins; creating a new Munin service thus takes a minimum of such boilerplate, allowing the programmer to focus on the business logic.

Service Types. The Munin Service layer defines the following service types (new types may be added later):

- `Display`: Display services are responsible for creating a graphics context on the peer so that visual output can be generated. Thus, most peers only have one active display service. Display service implementations are thus tied to a given graphics API.
- `Renderer`: Renderer services transform an abstract node in the distributed scene graph (Section 7) into graphical output on the peer's display. Renderers are therefore heavily reliant on the display service. Borrowing from the flexible rendering pipelines of IVTK [22], we allow several renderers to be active, each capable of interpreting particular scene graph nodes. This design also supports monolithic scene

graph nodes that can contain an entire visual representation, for example a single node representing a treemap. Such monolithic nodes are sometimes necessary for high-performance rendering.
- `Input`: Input services create the connection between input devices and the system. This involves transforming from the device's coordinate space into the global coordinate space based on the space configuration.
- `App`: Application services tie together a collection of services using business logic for a particular task.
- `Simulation`: Whereas other services are event-driven, simulation services have an active thread of execution for online processes such as animation.
- `Computation`: A computation service is one that operates on shared data to produce new data. This can be used to control how potentially expensive algorithms are distributed in the Munin space, e.g., to avoid overloading mobile devices.

Space Configuration. Munin maintains a global space configuration that lists all peers, their hardware capabilities (number of screens, resolution, input devices, etc), and their physical arrangements. This configuration is used when writing assembly specifications that map necessary services for an assembly onto actual peers in the Munin space. An assembly is thus tailored to a configuration.

The space configuration also contains definitions for one or several (2D) *surfaces*, each of which will hold a distributed scene graph in the Visualization layer. Peers select parts of surfaces for their own displays, which will control what is shown on that peer's screen. Since Munin is a pure peer-to-peer system, the space configuration is global to all peers. This is necessary to allow peers to define their own *area of interest* and determine whether any events or object updates will affect them (and therefore should be handled by them). While this design is somewhat limited in how display space can be stitched together, it is a simple model that can be easily configured by the user.

7 Munin: Visualization Layer

The top Munin layer is the Visualization layer, which brings together the lower layers to explicitly support building assemblies for ubiquitous visualization and analytics.

Shared Data Table. The relational data model has been shown to be a common data model for visualization [60], and many existing toolkits are based around this construct [22]. Munin accordingly defines a *shared relational data table* using the functionality of the Shared State layer. Tables can be published and subscribed to like normal objects, but they enforce a column-centric data model with a common schema. Table state is represented by rows in the table, with each column having the same data type. Shared tables can be queried (for searching for specific entries in the data), cascaded (for creating views of the data), and linked (for combining several tables).

Distributed Scene Graph. Each surface in the space configuration generates a scene graph that is distributed across the Munin space. Similar to most scene graphs, the Munin scene graph model is an instance of the Composite software design pattern, and is realized by Munin's shared objects. Taken in conjunction with the dynamic rendering service mechanism, this enables us to build component-based and truly reusable visual computing systems that separate the underlying data from the visual representation.

Furthermore, the fact that the scene graph is available to all services makes it possible to define cross-data instruments capable of manipulating all objects regardless of type. For example, similar to the VIGO model for multisurface computing [19], this allows us to define a pen instrument that can be used to scribble annotations on any visual representation. The renderer responsible for drawing the visual representation will automatically invoke the scribble renderer when it comes across a scribble node in the distributed scene graph.

Executing Assemblies. Finally, since Munin spaces generally consist of multiple networked devices, it can be painful to separately manage all of the services, configurations, and run scripts associated with each peer, one at a time. Instead, we provide the Munin LaunchPad as a peer-to-peer application frontend that runs on all connected peers and makes it simple to execute a Munin assembly. Executing an assembly will resolve all dependencies, download and upgrade all required services, and launch services in the correct order (for example, the display service must always be started before rendering services). The launcher also supports swapping plugins as well as shutting down and restarting the space.

8 Implementation

We have implemented Munin as a Java service platform that uses JGroups[5] for peer-to-peer multicast and the Java Simple Plugin Framework[6] for supporting dynamically-loaded services. The basic Munin distribution, including core services, is approximately 10,000 lines of Java code and can be distributed as a JAR file for easy use in a project. We enumerate the core services in the distribution below.

Display and Rendering Services. We have implemented display services for both computers and mobile devices:
- *A Java2D display service* using the Piccolo2D[7] vector graphics toolkit (reference implementation);
- *A JavaFX*[8] *experimental display service* that integrates Munin with web-based visualizations; and
- *A basic Android display service* that uses the Android SDK Canvas and Drawable APIs for mobile graphics.

Unfortunately, because no port of JGroups to iOS currently exists, we have not yet created display service implementations for Apple iPhone and iPad devices. Now that the Oracle ADF mobile framework allows Java development on iOS, this is a high priority item for the future.

Rendering services transform abstract scene graph nodes into graphical output using the display service. This means

5. http://www.jgroups.org/
6. http://code.google.com/p/jspf/
7. http://www.piccolo2d.org/
8. http://www.javafx.com/

that rendering services are closely coupled with display services in the current design. Munin puts no restrictions on the scene graph representation; in our reference display and rendering service implementations, we therefore use the SVG 1.1 (Scalable Vector Graphics) W3C standard.[9] In other words, SVG elements are implemented as Munin shared objects that maintain geometry, visual appearance, and hierarchy as properties. For SVG groups (`<svg:g>`) and other SVG elements that maintain references to other parts of the scene graph, we store the UUIDs of those objects as property values. To avoid dependency problems due to asynchronous updates of the Munin shared space, each element stores its parent (instead of vice versa).

Finally, the flexible Munin rendering service architecture means that an assembly can define new types of scene graph nodes and provide renderer implementations capable of interpreting them. We have used this in one of our examples (Section 9.2) to encapsulate Google Maps objects inside a Munin scene graph. Our reference SVG renderer serves as the fallback (an empty bounding box) for scene graph nodes that no other rendering service is capable of drawing.

Input Services. We have built some simple input services that an assembly can use to take advantage of the various input devices that a Munin space can be expected to have. For example, we provide both a mouse and a keyboard input service that listens for input on the local peer and distributes that input data to the shared event channel. Mouse coordinates are normalized based on the input area on the peer, allowing a pointer to traverse an entire surface. Managing keyboard focus is left to the assembly since we do not want to impose restrictions on the design; all peers will receive all keyboard events, and will have to determine which peer should handle them. There is also no restriction on the number of active services, allowing multiple mice or keyboards to active simultaneously.

Of course, one of the goals of Munin is to go beyond just mouse and keyboard input, and we provide several input services for this. First and foremost, we have a simple TUIO input service based on the reacTIVIsion Java client reference implementation which connects to a running TUIO server and listens to input events. We currently handle TUIO cursors, which typically arise from fingertop touches on multitouch surfaces, and actually store them not as events but as shared objects because they are persistent throughout the duration of a touch. Similarly, we also handle Kinect input through the TUIO interface, and we plan to support full skeletal input using OpenNI.

9 Application Examples

We here describe three visual applications we built using Munin. Other examples include a collaborative vector editor, a mobile sketch system for early design, and a simple multiplayer Tetris game.

9. http://www.w3.org/TR/SVG/

9.1 Multi-Device Collaborative Visual Search

One of our most sophisticated Munin assemblies to date is a collaborative visual search system designed for use in a ubiquitous visual computing space consisting of both a tabletop display serving as a shared and public view, and a set of mobile tablets serving as personal and private views (Figure 4). The idea behind this setup is that large tabletop displays are natural places for coupled work where participants are working in synchrony, whereas a mobile tablet has a more private form factor that suggests independent and uncoupled work. The product being refined by the collaborators is a query string used to filter search results, and we provide ways for participants to branch off from the shared view, explore data on their own using their tablet, and then merge back changes to the shared visual query on the tabletop.

Fig. 4. Collaborative visual search for real estate.

This visual search assembly has been implemented as a set of services running on the tabletop, and a dedicated app on the Android tablets. The tabletop runs the Java2D display service, the default SVG renderer for general shapes, and an OpenStreetMap renderer that implements the zoomable and pannable map in the center of the shared view. The tabletop also handles TUIO events internally. The geospatial dataset is distributed using Munin's shared table construct, whereas the query string is a shared object that is modified by both the tabletop as well as the tablets when users change filter settings.

Fig. 5. PolyZoom [64] running on a 3×2 LCD display wall. Android devices are used to interact with the map.

9.2 Distributed PolyZoom

PolyZoom [64] lets users create hierarchies of zoom regions in 2D spaces, which is particularly useful for navigating

in geographic maps. Instead of replacing a view of, say, France with a zoomed-in view of Paris, the Paris view becomes a child to the France view. However, the technique has so far only been deployed on single-user personal computers. We were interested in applying the technique to large-scale display environments, and to utilize its multifocus nature to allow collaborators to navigate in a map without interfering with each other.

Figure 5 shows a photograph of our Distributed PolyZoom assembly that we implemented using Munin. The three computers running the LCD display wall utilize the experimental JavaFX display service in order to natively access the Google Maps JavaScript API in Munin itself. The PolyZoom hierarchy is represented in the Munin scene graph with each Google Maps view becoming a special node type. A special JavaFX renderering service interprets the map parameters (longitude, latitude, and dimensions) embedded in these special nodes and fetches the corresponding graphics using the Google Maps API. Similarly, the Android app uses the Munin for Android SDK to connect to the Munin space and access the scene graph.

This application was built by a single programmer over a period of one month, during which he was first trained to use Munin framework through the documentation and demo applications that showed how to create and manage shared objects, events, and services. After this, the programmer created a display service using JavaFX to render the PolyZoom tree and a data structure for storing the tree data on the shared memory. This also demonstrates the flexibility of the Munin system; Munin does not require a specific graphics library, but instead the programmer could pick JavaFX for rendering because it was most suitable.

The map parameters for each map node in the PolyZoom tree are maintained in a shared data object that is created when an application user clicks on its parent node to zoom. The Android application displays all the selected regions on a root map of PolyZoom tree created using the Android maps API, thus giving an overview of the PolyZoom tree. The Android application also provides selection tools to select rectangular regions in the map, in order to create new nodes in the map hierarchy. The application user can also use the pinch-to-zoom feature provided by Android Google maps API to select small regions in the map. The application allows concurrent use by multiple users and handles their requests by their temporal order.

9.3 Shard: A Distributed Media Player

Shard is a distributed media player designed for decoupling media and playback as well as supporting multiple output and input surfaces (Figure 6). The player uses Munin as a backend and runs as a single service on all connected devices to enable high-performance rendering of HD-quality digital media. Each peer creates a player object in the Munin shared memory that all other peers automatically subscribe to, causing them to be notified when its state changes. All player objects contain the same data: the state of the player, the media URL, and the current position in the media being played back. Peers are only responsible for modifying their own object state, but will automatically respond to changes in other player objects (by matching state changes). This means that any participating device can change its own state, e.g., from playing to stopped, causing a cascade of changes in other peers.

Fig. 6. The Shard distributed media player displaying the Creative Commons movie Sintel (http://www.sintel.org/) on a tiled-LCD display wall (three computers).

Our implementation uses the vlcj[10] bindings for digital media decoding and playback. For situations when media is limited to a single device (such as a DVD or Bluray disc), Shard transparently creates an ad hoc RTP (Real-time Transport Protocol) server that streams the media from the source to all other peers using Munin for synchronization.

An Android remote was also built for Shard to control the playback. It maintains all the shared player objects created by the participating devices and gives the user options to pause, play, forward, and rewind the playback. If the media is present on the device, the app is also capable of playing the media locally (our device does not support RTP).

Shard can handle display surfaces built with many individual displays. Since Munin can handle creation and updation of player objects at real-time (discussed in Section 11.1), the bottleneck for this application lies at the streaming capabilities of the RTP server, which depends on video quality and network traffic, not in Munin itself.

10 Developing Using Munin

To demonstrate how to build a ubiquitous analytics space using Munin, we here discuss the steps involved in creating MultiDim, the multidimensional visualization assembly shown in Figure 1. The purpose of MultiDim is to allow multiple collaborators, each equipped with a personal mobile phone or tablet, to use a large display environment for social data analysis by uploading datasets, creating multidimensional visualizations, and annotating them.

Building a new assembly is done as follows:

- **Defining the physical setup:** even if Munin spaces are designed to work on any hardware ensemble, the physical setup is an important aspect of designing the visualization and interaction;

10. http://www.capricasoftware.co.uk/vlcj/

- **Selecting existing services:** Munin provides a rich (and growing) ecology of reusable services, and thus selecting which existing service to use is a vital step in building an assembly;
- **Creating new services:** this is the coding part of creating a new Munin assembly—at the very least, new assemblies tend to create an App service, and may need to define additional services;
- **Assembly specification:** the assembly XML specification determines which services will run on which peers in the Munin space; and finally
- **Launching the assembly:** we use the Munin Launchpad to run the assembly in the distributed environment.

In the following subsections, we go into full detail on how to perform each of these steps for MultiDim.

10.1 Step 1: Physical Setup

We will construct an assembly that will use a single 2D surface that is realized by several wall-mounted displays for output (Figure 7). The surface will also comprise a tabletop display that resides in the lower part of the surface (consistent with its physical placement in front of the wall-mounted displays). Android devices will allow for uploading datasets as well as moving a cursor (one per device) anywhere on the surface displayed using the display environment, but will render no output. This is to ensure that all visuals are drawn on shared displays.

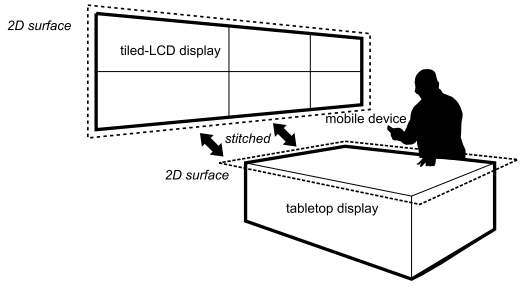


Fig. 7. Physical setup for the multidimensional visualization assembly. The vertical surface on the wall display and the horizontal on the tabletop form one single virtual 2D surface on which panels can be displayed.

10.2 Step 2: Selecting Existing Services

All Munin assemblies consist of an ensemble of interdependent services that use the Munin shared space and event channel to communicate. Many services provide general functionality that many different assemblies can reuse and are therefore part of the Munin standard distribution. Therefore, it often makes sense to select among existing services to save time and effort in implementing a new assembly.

For MultiDim, we choose to use the standard Java2D display service and the SVG renderer. These services provide basic vector graphics specified using the SVG 1.1 standard that we will use to create our visualizations. We will also use the ZoomPanner, an app service that transforms cursor events into transformations applied to the root scene graph node. This allows any cursor input service (such as a mouse or Kinect gesture detector) to zoom and pan the scene.

10.3 Step 3: Creating New Services

Several new services must be created for this assembly. Because Munin services can be reused in other assemblies, it often makes sense to define each service to be very focused instead of performing multiple different services. For MultiDim, we create three separate services:

- **MultiDim:** The main app service that loads data from a local data source and creates the corresponding SVG visualizations. The service will listen to cursor events for specifying the position and dimensions of new visualization panels, and will then create the selected visualization in that space. Listing 1 shows a Java snippet adding the marks for a scatterplot visualization to the scene graph. Note that changes to shared objects only take effect (i.e., update the local space as well as transmitted across the P2P channel) after they have been committed (`pt.commit()`).
- **Scribble:** A handwriting app service that allows users to annotate the shared space. The service simply creates an SVG polyline based on cursor input using a chosen color and stroke width.
- **ZoomPanner:** A service that allows the users to zoom or pan a display surface. The service changes the world transform for the surface based on cursor input.

When several app services that all listen to event input are running in the same Munin space, arbitration is needed to prevent several services from acting on the same event. This is because Munin due to its distributed nature does not have a way to consume an event and stop its propagation. For example, if MultiDim, Scribble, and ZoomPanner all handled the same cursor input event, the user would be defining a visualization panel, scribbling, and panning using the same action. Munin currently has no standardized way to support arbitration, so we use function keys (F1-F12) to swap between active app services.

```
for (int i = 0; i < table.getRowCount(); i++) {
    float x = (xCol.getVal(i) - xCol.getMin())/xd;
    float y = (yCol.getVal(i) - yCol.getMin())/yd;
    float c = (cCol.getVal(i) - cCol.getMin())/cd;
    Color color = new Color(c, 0.0f, 0.0f, 0.2f);
    SharedObject pt = getSpace().createObject();
    SceneGraph.createCircle(pt, x * w, y * h, 5);
    SceneGraph.setFillColor(pt, color);
    SceneGraph.setStrokeColor(pt, Color.lightGray);
    SceneGraph.addChild(panel, pt);
    pt.commit();
}
```

Listing 1. Adding marks (SVG circles) to a scatterplot based on data columns for X, Y, and color coordinates.

Mobile devices do not run services, but are instead implemented as Android apps. Each Android device will run a MULTIDIM app built using the Munin for Android API. This app is very simple: it will use the space configuration to render the virtual surface on the mobile device's screen with labeled rectangles to show the individual displays and

their positions. Whenever the user interacts with the screen, the touch events will be translated into shared events that are then propagated in the Munin event channel. Listing 2 shows a code snippet from the app's touch event handler.

```
public boolean onTouchEvent(MotionEvent e) {
   SharedEvent touch = getSpace().createEvent();
   touch.putString("input", "cursor");
   touch.putString("peer", peer.getId());
   touch.putDouble("x", convertToWorldX(e.getX());
   touch.putDouble("y", convertToWorldY(e.getY());
   if (e.getAction()==MotionEvent.ACTION_DOWN)
      touch.putBoolean("down", true);
   else if (e.getAction()==MotionEvent.ACTION_MOVE)
      touch.putBoolean("move", true);
   else if (e.getAction()==MotionEvent.ACTION_UP)
      touch.putBoolean("up", true);
   touch.send();
   return true;
}
```

Listing 2. Converting touch into Munin cursor events.

Finally, the tabletop device is special because it also supports input from multitouch interaction, unlike the wall-mounted displays. We use our TUIO service to handle input events from tabletop interaction.

10.4 Step 4: Assembly Specification

The assembly specification is an XML file that describes which peers in a Munin space will execute which services (and in which order). Each peer has a copy of this file. It is intimately tied to a specific space configuration, and thus to a physical Munin space, and must therefore be modified if the assembly is to run in a different physical space.

For the MultiDim assembly, we configure each of the four computers involved in the physical setup to run two copies of the reference Java2D display service (one for each screen). Furthermore, each computer will run a single SVG renderer; the renderer is not tied to a specific display service, so it can render scene graphs for both display services running on each computer. Since both ZoomPanner and Scribble are app services that use the shared event channel, we only need to run one instance of each on one of the four computers. Similarly, the MultiDim service itself really only needs one instance, but can be run on any computer from where to find the dataset to visualize.

Note that Android devices are not part of the assembly because they are not launched together with the rest of the space; instead, they launch when the user starts the app. Also, Android devices are not part of the virtual 2D surface specified by the space configuration.

10.5 Step 5: Executing the Assembly

Assemblies are executed using the Munin LaunchPad. First, instances of the LaunchPad must be started on all participating peers, and the assembly specification must be replicated (we achieve this with a shared file system, but a better approach would be to automatically distribute these XML files using Munin). Executing the "Launch" command on one of the instances will launch the assembly on all participating peers.

Fig. 8. Android tablet interacting with MULTIDIM.

Figure 8 shows the MultiDim assembly being used with a tablet, and Figure 1 shows the entire board. The entire Java source code required for building this assembly is less than 500 lines (mostly programmatic SVG generation), plus an additional 50 lines of XML.

11 EVALUATION

Beyond our example applications above, we also validate Munin using performance testing as well as an informal questionnaire distributed to Munin end-user programmers. This section summarizes our findings.

11.1 Performance Results

A peer-to-peer architecture such as Munin typically benefits from better performance than client/server architectures due to the removal of the central server as a bottleneck for all communication. We refer to existing performance tests conducted for the JGroups library for raw network performance.[11] Instead, the purpose of our performance testing for Munin was to give insight into its efficiency for a typical visualization setting, including both data transfer and rendering. We therefore chose to measure the delays involved in following Stefik et al.'s design guideline "What you see is what I see" [16] for multiuser interfaces.

Table 2 shows performance results for a simple Munin setup consisting of two desktop computers connected using a gigabit Ethernet connection (same switch, same subnet). The stimulus was adding a number of SVG rectangles at random locations to the Munin scene graph, simulating the creation of a large-scale 2D scatterplot. Our test measured the time for data transfer as well as rendering from the moment the originating peer sent the first update to the time the destination peer received and rendered the last. The measured time is in the order of milliseconds for 50 entities, which was a typical number for our application examples. These informal findings indicate that Munin scales well to most standard visualization tasks. Note that the results presented here correspond to creation of all shared objects at once but in case of our application examples the programmers observed that they never had to create or update

11. http://www.jgroups.org/performance.html

more than 10 shared objects at once. This led to a real-time performance of the distributed system. In the case of Shard, RTP streaming seemed to cause a barely perceivable delay for high definition (1900×1080 HD) media.

TABLE 2
Performance data for replicating and rendering SVG rectangles in the Munin distributed scene graph.

Entities	Total Size (Mb)	Transfer + Rendering (sec)
1,000	0.24	2.05
25,000	6.1	3.37
50,000	12.2	8.83
100,000	24.4	15.98
200,000	48.4	35.02

11.2 End-User Feedback

Munin is currently publicly available as Open Source on GitHub[12], and there is a small number of end-user programmers that have used the framework to build visualization assemblies. For example, both the Shard and Distributed PolyZoom examples were built by end-user programmers. To collect informal feedback on the framework, we administered a questionnaire asking them about their experience.

All respondents were competent Java programmers (3.5 mean on a 5-point Likert scale), and the results were uniformly positive. The participants felt that they were able to become proficient with the framework (defined as being able to build assemblies on their own) in as little as a few days and up to a week or two. While all respondents conceded that distributed programming was difficult, they felt that Munin's conceptual model made the task easier in understanding how events and data are replicated across the space. They thought that once they had grasped the idea, Munin provided some very challenging functionality "for free," including sharing and stitching screens, drawing from existing services, and distributing events across the space.

12 Design Implications

We believe that ubiquitous analytics, an exciting new area of research, is currently hindered by the lack of standard frameworks for building visual computing software that scale seamlessly to multiple devices, displays, and surfaces. Munin is one such framework that can be used to build distributed visual computing enviroments, but it is important to emphasize that there is significant opportunity for developing other toolkits with differing design rationale and architecture. In this section, we discuss some of the strengths and weaknesses of the Munin toolkit.

We chose Java as the programming language for Munin to achieve some of our design goals, and we utilize existing communication modules such as JGroups, and graphics modules such as Piccolo2D, JavaFX, and native Android graphics APIs for achieving the envisioned shared state, shared service, and visualization layer objectives. The choice of programming language should ideally make it out-of-box compatible with a wide range of desktop platforms and also mobile platforms such as Android and iOS. However, some of the dependencies of Munin are yet to be ported to the iOS platform and this remains to be corrected in near future. Munin can also be integrated with software toolkits written in general-purpose languages such as C and C++ through standard interfacing options such as Java Native Interface (JNI). This does require rewriting the target software toolkits to utilize Munin's service-oriented model. It is also worth mentioning that while Java is platform independent and does not require rebuilding the code to every platform, mobile platforms do not provide complete support to typical Java rendering libraries. Therefore, we have created a separate rendering service for Android, and the same is required for iOS.

We opted for a peer-to-peer architecture over a more traditional client/server one, and this had significant repercussions on our design. Based on our previous experience building client/server systems, they are typically relatively simple to implement, provide a straightforward location for arbitrating between clients and executing exclusive logic, and are easy to understand and learn for the application programmer. For example, most current web-based collaborative applications (WebEx, Adobe Connect, etc) use a client/server architecture where the server handles events and tasks. On the other hand, P2P systems require event and task management models that work with distributed logic and are capable of managing conflicts, synchronization, and scalability issues in each individual peer. In our P2P design, each peer must independently determine whether a particular event or object falls in their area of concern, and the distribution of services across peers is asymmetric to take advantage of device-specific hardware capabilities.

On the other hand, the centralization in a client/server system may cause issues for scalability, robustness, security, and trust. In contrast, the Munin P2P architecture is more robust, more fault-tolerant, and more loosely coupled than a client/server design. Furthermore, even if peer-to-peer architectures can be difficult to understand and learn, we are convinced that the data-driven and service-oriented programming model we use reduces much of this complexity. At the same time, it is clear that a distributed visualization executing on several computers simultaneously will be more complex to develop than a single-computer one. Infact, typical issues such as display synchronization that can be easily handled or even eliminated through client/server model may need additional program logic in P2P architecture through acknowledgement procedures. Our anecdotal findings point in the right direction, but more evaluation with end-user programmers is needed.

Munin is designed for co-located synchronous settings, but there are many situations where users in one ubiquitous visual computing space will want to connect to users in another location to collaborate remotely on a particular problem. Our implementation currently does not support this scenario, but one approach may be to integrate Munin with the Hugin [18] framework for remote visual computing by simply integrating Hugin clients (Hugin uses a clien-

12. https://github.com/karthikbadam/Munin

t/server architecture) as Munin peers. Munin objects that should be exported remotely could be tagged with reserved Hugin properties for permissions and ownership.

Our shared data model in Munin has no encapsulation, access control, or type checking. This makes for rapid and effortless development, but may cause bugs that are difficult to track down if a service changes a shared object in an erratic way. One approach may be to adopt the Governor pattern proposed by Klokmose and Beaudouin-Lafon in the VIGO model [19], where all shared objects may be associated with an entity that can determine whether a state change is legal, and also execute logic associated with the object if it changes. However, this is left for future work.

13 Conclusion and Future Work

We have presented the Munin framework for building distributed software in special-purpose multi-surface environments that we call *ubiquitous analytics and visualization spaces*. Beyond the basic functionality of existing middlewares for multi-surface environments, Munin is particularly suitable for high-performance visual computing due to (1) a structured data table for shared state; (2) a distributed scene graph that delegates rendering to each device; and (3) a data-driven programming model where cross-cutting services manage input, output, and computation. We validated Munin using several applications built using the framework, an in-depth example describing its use, and both performance results as well as qualitative feedback.

In the future, we plan on extending the collection of display, rendering, input management, and computation services, as well as adapt the framework to additional platforms. We also envision adding significantly to the vocabulary of visualization techniques that Munin supports.

References

[1] B. Lee, P. Isenberg, N. H. Riche, and S. Carpendale, "Beyond mouse and keyboard: Expanding design considerations for information visualization interactions," *IEEE Transactions of Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2689–2698, 2012.

[2] F. Guimbretière, M. C. Stone, and T. Winograd, "Fluid interaction with high-resolution wall-size displays," in *Proceedings ACM Symposium on User Interface Software & Technology*, 2001, pp. 21–30.

[3] P. Wellner, "Interacting with paper on the DigitalDesk," *Communications of the ACM*, vol. 36, no. 7, pp. 86–96, Jul. 1993.

[4] H.-C. Jetter, M. Zöllner, J. Gerken, and H. Reiterer, "Design and implementation of post-WIMP distributed user interfaces with ZOIL," *International Journal of Human-Computer Interaction*, vol. 28, no. 11, pp. 737–747, 2012.

[5] N. A. Streitz, J. Geissler, T. Holmer, S. Konomi, C. Müller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, and R. Steinmetz, "i-LAND: An interactive landscape for creativity and innovation," in *Proc. ACM Conference on Human Factors in Computing Systems*, 1999, pp. 120–127.

[6] T. Gjerlufsen, C. N. Klokmose, J. Eagan, C. Pillias, and M. Beaudouin-Lafon, "Shared substance: developing flexible multi-surface applications," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2011, pp. 3383–3392.

[7] P. Isenberg, A. Tang, and M. S. T. Carpendale, "An exploratory study of visual information analysis," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2008, pp. 1217–1226.

[8] A. van Dam, "Post-WIMP user interfaces," *Communications of the ACM*, vol. 40, no. 2, pp. 63–67, Feb. 1997.

[9] M. Weiser, "The computer for the twenty-first century," *Scientific American*, vol. 3, no. 265, pp. 94–104, Sep. 1991.

[10] N. Elmqvist and P. Irani, "Ubiquitous analytics: Interacting with big data anywhere, anytime," *IEEE Computer*, vol. 46, no. 4, pp. 86–89, 2013.

[11] P. Isenberg, N. Elmqvist, J. Scholtz, D. Cernea, K.-L. Ma, and H. Hagen, "Collaborative visualization: definition, challenges, and research agenda," *Information Visualization*, vol. 10, no. 4, pp. 310–326, 2011.

[12] R. J. K. Jacob, A. Girouard, L. M. Hirshfield, M. S. Horn, O. Shaer, E. T. Solovey, and J. Zigelbaum, "Reality-based interaction: a framework for post-WIMP interfaces," in *Proceedings ACM Conference on Human Factors in Computing Systems*, 2008, pp. 201–210.

[13] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A scalable parallel rendering framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436–452, 2009.

[14] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski, "Chromium: a stream-processing framework for interactive rendering on clusters," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 693–702, Jul. 2002.

[15] R. A. Bolt, ""Put-That-There": voice and gesture at the graphics interface," *Computer Graphics*, vol. 14, no. 3, pp. 262–270, 1980.

[16] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. G. Tatar, "WYSIWIS revised: Early experiences with multiuser interfaces," *ACM Transactions on Office Information Systems*, vol. 5, no. 2, pp. 147–167, 1987.

[17] P. Isenberg and S. Carpendale, "Interactive tree comparison for co-located collaborative information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1232–1239, 2007.

[18] K. Kim, W. Javed, C. Williams, N. Elmqvist, and P. Irani, "Hugin: A framework for awareness and coordination in mixed-presence collaborative information visualization," in *Proceedings ACM Conference on Interactive Tabletops and Surfaces*, 2010, pp. 231–240.

[19] C. N. Klokmose and M. Beaudouin-Lafon, "VIGO: instrumental interaction in multi-surface environments," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2009, pp. 869–878.

[20] J. Melchior, D. Grolaux, J. Vanderdonckt, and P. V. Roy, "A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications," in *Proceedings of the ACM Symposium on Engineering Interactive Computing System*, 2009, pp. 69–78.

[21] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-driven documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 6, pp. 2301–2309, 2011.

[22] J.-D. Fekete, "The InfoVis Toolkit," in *Proceedings of the IEEE Symposium on Information Visualization*, 2004, pp. 167–174.

[23] P. Isenberg and D. Fisher, "Collaborative brushing and linking for co-located visual analytics of document collections," *Computer Graphics Forum*, vol. 28, no. 3, pp. 1031–1038, 2009.

[24] S. Panëels and J. C. Roberts, "Review of designs for haptic data visualization," *IEEE Transactions on Haptics*, vol. 3, no. 2, pp. 119–137, 2010.

[25] M. Tobiasz, P. Isenberg, and S. Carpendale, "Lark: Coordinating co-located collaboration with information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1065–1072, 2009.

[26] C. Hansen, T. Crockett, and S. Whitman, "Guest Editor's introduction: Parallel rendering," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 2, no. 2, pp. 7–7, 1994.

[27] T. Ni, G. S. Schmidt, O. G. Staadt, M. A. Livingston, R. Ball, and R. May, "A survey of large high-resolution display technologies, techniques, and applications," in *Proceedings of the IEEE Conference on Virtual Reality*, 2006, pp. 223–236.

[28] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A virtual platform for virtual reality application development," in *Proceedings of the IEEE Conference on Virtual Reality*, 2001, pp. 89–96.

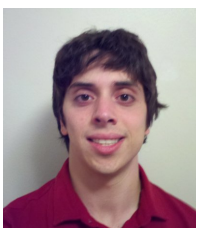
[29] V. Anupam, C. Bajaj, D. Schikore, and M. Schikore, "Distributed and collaborative visualization," *Computer*, vol. 27, no. 7, pp. 37–43, 1994.

[30] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, Jul. 1994.

[31] K. W. Brodlie, D. A. Duce, J. R. Gallop, J. P. R. B. Walton, and J. D. Wood, "Distributed and collaborative visualization," *Computer Graphics Forum*, vol. 23, no. 2, pp. 223–251, 2004.

[32] A. Ebert, S. Thelen, P.-S. Olech, J. Meyer, and H. Hagen, "Tiled++: An enhanced tiled hi-res display wall," *IEEE Trans. on Visualization and Computer Graphics*, vol. 16, no. 1, pp. 120–132, 2010.
[33] B. Jeong, L. Renambot, R. Jagodic, R. Singh, J. Aguilera, A. Johnson, and J. Leigh, "High-performance dynamic graphics streaming for scalable adaptive graphics environment," in *Proceedings of the ACM/IEEE Supercomputing Conference*, 2006, pp. 24–33.
[34] B. Schaeffer, P. Brinkmann, G. Francis, C. Goudeseune, J. Crowell, and H. Kaczmarski, "Myriad: scalable vr via peer-to-peer connectivity, pc clustering, and transient inconsistency," *Computer Animation and Virtual Worlds*, vol. 18, no. 1, pp. 1–17, 2007.
[35] B. Shen, J. Guo, and P. Chen, "A survey of P2P virtual world infrastructure," in *Proceedings of IEEE Conference on e-Business Engineering*, 2012, pp. 296–303.
[36] S. Feiner and A. Shamash, "Hybrid user interfaces: Breeding virtually bigger interfaces for physically smaller computers," in *Proceedings ACM Symposium on User Interface Software & Technology*, 1991, pp. 9–17.
[37] A. Fox, B. Johanson, P. Hanrahan, and T. Winograd, "Integrating information appliances into an interactive workspace," *IEEE Computer Graphics and Applications*, vol. 20, no. 3, pp. 54–65, 2000.
[38] R. Raskar, G. Welch, M. Cutts, A. Lake, L. Stesin, and H. Fuchs, "The office of the future: A unified approach to image-based modeling and spatially immersive displays," *Computer Graphics*, vol. 32, pp. 179–188, Aug. 1998.
[39] N. Elmqvist, "Distributed user interfaces: State of the art," in *Distributed User Interfaces: Designing Interfaces for the Distributed Ecosystem*, J. A. Gallud, R. Tesoriero, and V. M. R. Penichet, Eds. Springer, 2011.
[40] J. Coutaz, L. Balme, C. Lachenal, and N. Barralon, "Software infrastructure for distributed migratable user interfaces," in *Proceedings of the UbiHCISys Workshop on UbiComp*, 2003.
[41] P. Tandler, "Software infrastructure for ubiquitous computing environments: Supporting synchronous collaboration with heterogeneous devices," *LNCS*, vol. 2201, pp. 96–115, 2001.
[42] M. Modahl, I. Bagrak, M. Wolenetz, P. W. Hutto, and U. Ramachandran, "MediaBroker: An architecture for pervasive computing," in *Proceedings of the IEEE Conference on Pervasive Computing*, 2004, pp. 253–262.
[43] N. Marquardt, R. Diaz-Marino, S. Boring, and S. Greenberg, "The proximity toolkit: prototyping proxemic interactions in ubiquitous computing ecologies," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, 2011, pp. 315–326.
[44] E. Pietriga, S. Huot, M. Nancel, and R. Primet, "Rapid development of user interfaces on cluster-driven wall displays with jBricks," in *Proceedings of the ACM Symposium on Engineering Interactive Computing System*, 2011, pp. 185–190.
[45] X. Bi and R. Balakrishnan, "Comparing usage of a large high-resolution display to single or dual desktop displays for daily work," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2009, pp. 1005–1014.
[46] A. Endert, L. Bradel, J. Zeitz, C. Andrews, and C. North, "Designing large high-resolution display workspaces," in *Proceedings of the ACM Conference on Advanced Visual Interfaces*, 2012, pp. 58–65.
[47] K. Moreland, "Redirecting research in large-format displays for visualization," in *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2012, pp. 91–95.
[48] H. Zhao, C. Plaisant, B. Shneiderman, and J. Lazar, "Data sonification for users with visual impairment: A case study with georeferenced data," *ACM Transactions on Computer-Human Interaction*, vol. 15, no. 1, 2008.
[49] A. Wu, "Tangible visualization," in *Proceedings of the Conference on Tangible and Embedded Interaction*, 2010, pp. 317–318.
[50] D. A. Bowman, C. North, J. Chen, N. F. Polys, P. S. Pyla, and U. Yilmaz, "Information-rich virtual environments: theory, tools, and research agenda," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, 2003, pp. 81–90.
[51] L. Chittaro, "Visualizing information on mobile devices," *IEEE Computer*, vol. 39, no. 3, pp. 40–45, 2006.
[52] S. Kim, Y. Jang, A. Mellema, D. S. Ebert, and T. W. Collins, "Visual analytics on mobile devices for emergency response," in *Proceedings of the IEEE Symposium on Visual Analytics Science and Technology*, 2007, pp. 35–42.
[53] P. Pombinho, "Information visualization on mobile environments," in *Proceedings of the ACM Conference on Human-Computer Interaction with Mobile Devices and Services*, 2010, pp. 493–494.
[54] C. Andrews, A. Endert, and C. North, "Space to think: Large, high-resolution displays for sensemaking," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2010, pp. 55–64.
[55] M. Czerwinski, G. Smith, T. Regan, B. Meyers, G. Robertson, and G. Starkweather, "Toward characterizing the productivity benefits of very large displays," in *Proceedings of INTERACT*, 2003, pp. 9–16.
[56] Z. Liu, N. J. Nersessian, and J. T. Stasko, "Distributed cognition as a theoretical framework for information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1173–1180, 2008.
[57] E. Hutchins, *Cognition in the Wild*. MIT Press, 1995.
[58] A. Clark and D. Chalmers, "The extended mind," *Analysis*, vol. 58, no. 1, pp. 7–19, 1998.
[59] L. Shapiro, *Embodied Cognition*. New York, NY: Routledge, 2011.
[60] J. Heer and M. Agrawala, "Software design patterns for information visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 853–860, 2006.
[61] O. Kreylos, "Environment-independent VR development," in *Advances in Visual Computing*. Springer, 2008, pp. 901–912.
[62] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985.
[63] B. Johanson and A. Fox, "Extending tuplespaces for coordination in interactive workspaces," *Journal Systems and Software*, vol. 69, no. 3, pp. 243–266, 2004.
[64] W. Javed, S. Ghani, and N. Elmqvist, "PolyZoom: multiscale and multifocus exploration in 2D visual spaces," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2012, pp. 287–296.

Sriram Karthik Badam received his Bachelor of Technology in Computer Science and Engineering in 2012 from Indian Institute of Technology Hyderabad in India. He is a Ph.D. student in the School of Electrical and Computer Engineering at Purdue University in West Lafayette, IN, USA. His research interests include human-computer interaction, information visualization, and visual analytics. He is a student member of the IEEE.



Eli Fisher received the bachelor of computer engineering degree from Purdue University in Spring 2014. He now works for Microsoft Corporation.

Niklas Elmqvist received his Ph.D. in 2006 from Chalmers University of Technology in Göteborg, Sweden. He is an assistant professor in the School of Electrical and Computer Engineering at Purdue University in West Lafayette, IN, USA. He was previously a postdoctoral researcher at INRIA Saclay - Île-de-France located at Université Paris-Sud in Paris, France. He is a senior member of the IEEE and the IEEE Computer Society.