2012

Software architectural support for tangible user interfaces in distributed, heterogeneous computing environments

Cornelius Toole
Louisiana State University and Agricultural and Mechanical College

SOFTWARE ARCHITECTURAL SUPPORT FOR TANGIBLE USER INTERFACES IN DISTRIBUTED, HETEROGENEOUS COMPUTING ENVIRONMENTS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Cornelius Toole, Jr.
B.S. Computer Science, Jackson State University, 2003
M.S. Computer Science, Jackson State University, 2005
August 2012

Acknowledgments

Despite the solitary nature of doctoral studies, this dissertation would not have been possible without the support of my advisor, family, friends and colleagues. Thanks to my committee, Professors Brygg Ullmer, Gabrielle Allen and Bijaya Karki for advising me throughout this process.

A special thanks goes to my advisor, Professor Brygg Ullmer, for supporting my research at Louisiana State University. He was very instrumental in my coming to the University for doctoral studies and has shown remarkable zeal in supporting my research interests. Throughout this process, he has been very patient and understanding in affording me time and space to conduct my research. I will always remember his admonishments to make no small plans, as well as his challenge to project out into the future.

I would also like to acknowledge, Professor Gabrielle Allen. From our first meeting, she has been concerned with the best course of action for my career. Thank you for your mentorship and opportunities to apply my research.

Thanks to my wife, Kenitta, who supported me without fail. Thank you for all your patience and encouragement. Words cannot express my love and admiration. To my children Leslie and Cornelius Raymond, I love you so much and am so grateful that you allowed your daddy to be away so much during his studies.

To my collaborators in the Tangible Visualization Laboratory and the Center for Computation and Technology (CCT) Scientific Visualization Group, I am appreciative for all the lively discussions and debates, all-nighters, meals, laughs and hard work we have been able to share over the years. Thanks especially to Rajesh Sankaran, Andrei Hutanu, Christopher Branton, Jinghua Ge and Kexi Liu.

This work was supported by the Huel D. Perkins Doctoral Fellowship, The NSF/LSAMP Bridge to the Doctoral Program Fellowship, The Viz Tangibles Project (NSF MRI-0521559), CreativeIT (NSF IIS-0856065), Cybertools (NSF EPS-070149), the eaviv project (NSF OCI 0947825), the LIGO Outreach Tangibles Project (LA BoR LEQSF (2008-09)-TOO-LIGO Outreach), and the CCT.

Table of Contents

List of Tables

List of Figures

Abstract

This research focuses on tools that support the development of tangible interaction-based applications for distributed computing environments. Applications built with these tools are capable of utilizing heterogeneous resources for tangible interaction and can be reconfigured for different contexts with minimal code changes. Current trends in computing, especially in areas such as computational science, scientific visualization and computer supported collaborative work, foreshadow increasing complexity, distribution and remoteness of computation and data. These trends imply that tangible interface developers must address concerns of both tangible interaction design and networked distributed computing. In this dissertation, we present a software architecture that supports separation of these concerns. Additionally a tangibles-based software development toolkit based on this architecture is presented that enables the logic of elements within a tangible user interface to be mapped to configurations that vary in the number, type and location of resources within a given tangibles-based system.

Chapter 1

Introduction

This research concerns the development and deployment of tangible user interfaces within computing environments with distributed, heterogeneous resources. More specifically, this research focuses on the intersection of tangible user interfaces, distributed computing and software architecture. Tangible user interfaces (TUIs) give physical form to digital information [IU10]. TUIs are valuable as an approach to user interfaces though enabling a user to apply knowledge about engaging the physical world she has already acquired toward engaging the digital world. The field of distributed computing is concerned with the study, design and development systems composed of nodes from multiple address spaces [Gof03]. A strength of distributed systems is that they enable remote access to remote resources (e.g. people, physical things, data and computation) [Dix08].

In the literature we find several examples of TUI-based applications that employ distributed resources of some type [BID98, MG07, KB07a, BRSB03]. Such systems inherit many of the benefits of tangibles supported by only local resources as well as the advantages of distributed systems. Contemporary computing trends (circa early 2012) also point to a future in which the computation of interest to user is spread across space and time over diverse resources. This can be observed in the proliferation of devices with embedded processing, networking connectivity and interactive capability as well the large-scale deployment of Web-based infrastructure to support business and consumer software applications (i.e. cloud computing[BYV+09]). Given these trends, tangibles in the contexts with distributed, heterogeneous computing resources also inherit the challenges and limitations of distributed systems. The limitations and challenges of distributed computing are captured by Deutsch and Gosling's *Eight Fallacies of Distributed Computing* [DG97]. The limitations and challenges of designing, developing and deploying tangibles remain as described by Shaer et al. [SJ09, SH10] (e.g. designing interplay of virtual and physical, lack of standard input and output devices, crossing disciplinary boundaries, etc.).

One implication of the intersection of TUIs and distributed computing is stakeholder diversity. These stakeholder include end-users, domain application developers, TUI developers and designers, TUI tools

developers, and developers of infrastructure for computation and networking. Each stakeholder class faces problems of distributed, heterogeneous tangible interaction unique to their area of expertise or interest.

End-users One end-user problem of distributed tangible interaction revolves around discoverability of affordances and functionality.

Domain Application Developers A domain application developer is responsible for developing or adopting applications and services to carry the computational task relevant to the domain for a class of activities. Ill-defined system specifications and requirements integration of domain-specific application components with TUI elements is a problem facing domain application developers.

TUI designer/developer TUI designers/developers are responsible for the composition of TUI elements to support the physically-linked invocation of software actions. This class of distributed TUI stakeholder faces problems such as: the lack of expertise on technical requirements for distributed interaction; the lack of abstractions for application integration of novel tangible interaction techniques; technical constraints in terms of operating system, programming language or application platform; abundance of choice of technologies for implementing TUIs; and the lack of tools for the design, development and deployment of TUIs for distributed, heterogeneous computing contexts.

TUI Tools Maker This class of stakeholder provides tangible interaction techniques to other distributed TUI stakeholders. They may face problems such as: ill-defined system specification and requirements for systems in which tangibles are to be integrated, both for novel and legacy applications; lack of expertise or affordability of focus upon technical concerns of distributed computing and communication infrastructure; and complexity as a result of the abundance of choice of technologies for distributed computing and tangible interaction (e.g. protocol, services, middleware, application frameworks, interaction hardware toolkit, etc.).

Distributed Interaction Infrastructure Developer This stakeholder provides the computational and communication infrastructure to support distributed, heterogeneous tangibles. In many cases, this infrastructure is not developed in consideration of tangible interaction. Some exceptions to reality-based interaction (RBI [JGH+08]) approaches like tangibles include the work of Ballagas, Holleis, Johanson and Fox, Kaltenbrunner, Kobayashi, Kumpf, and Marquardt and Greenberg [BRSB03, Hol09, JF04, KBBC05, Kum09, MG07]. This points to the issue of HCI infrastructure raised by Edwards et al. [ENP10]. A key problem facing infrastructure developers in distributed tangible interaction

is ill-defined specifications and requirements for infrastructure to support distributed, heterogeneous tangible interaction.

This enumeration of distributed, heterogeneous tangible interaction stakeholders is likely not comprehensive; and these roles overlap, as has been the case with the author. But this approach to examining the problems of distributed, heterogeneous tangible interaction does highlight the need for a higher-level view of the processes of designing, developing and deploying tangibles.

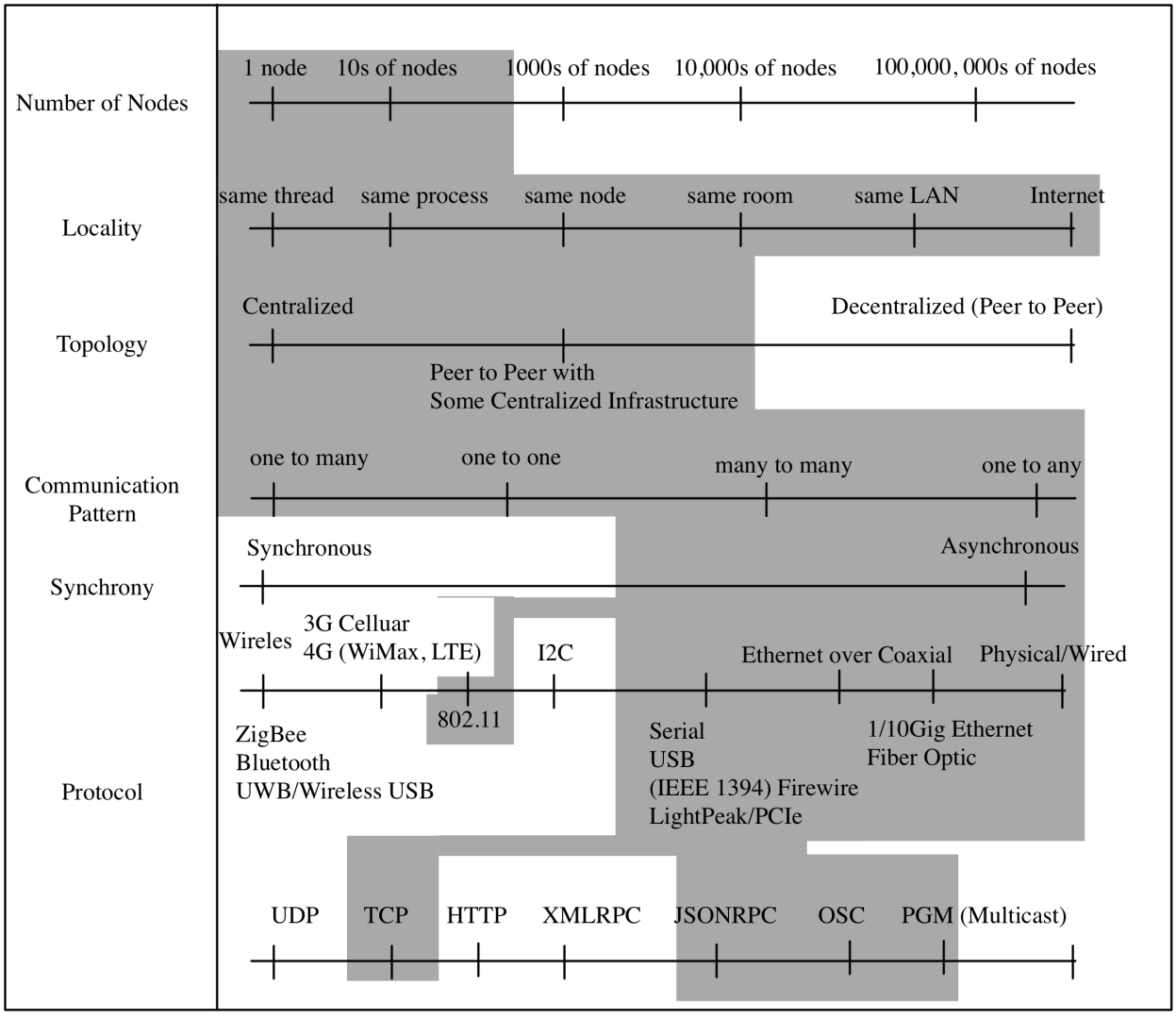

FIGURE 1.1: Design space of distributed systems for tangible Interfaces. The gray region represents areas within the design space that have been addressed by this dissertation's research.

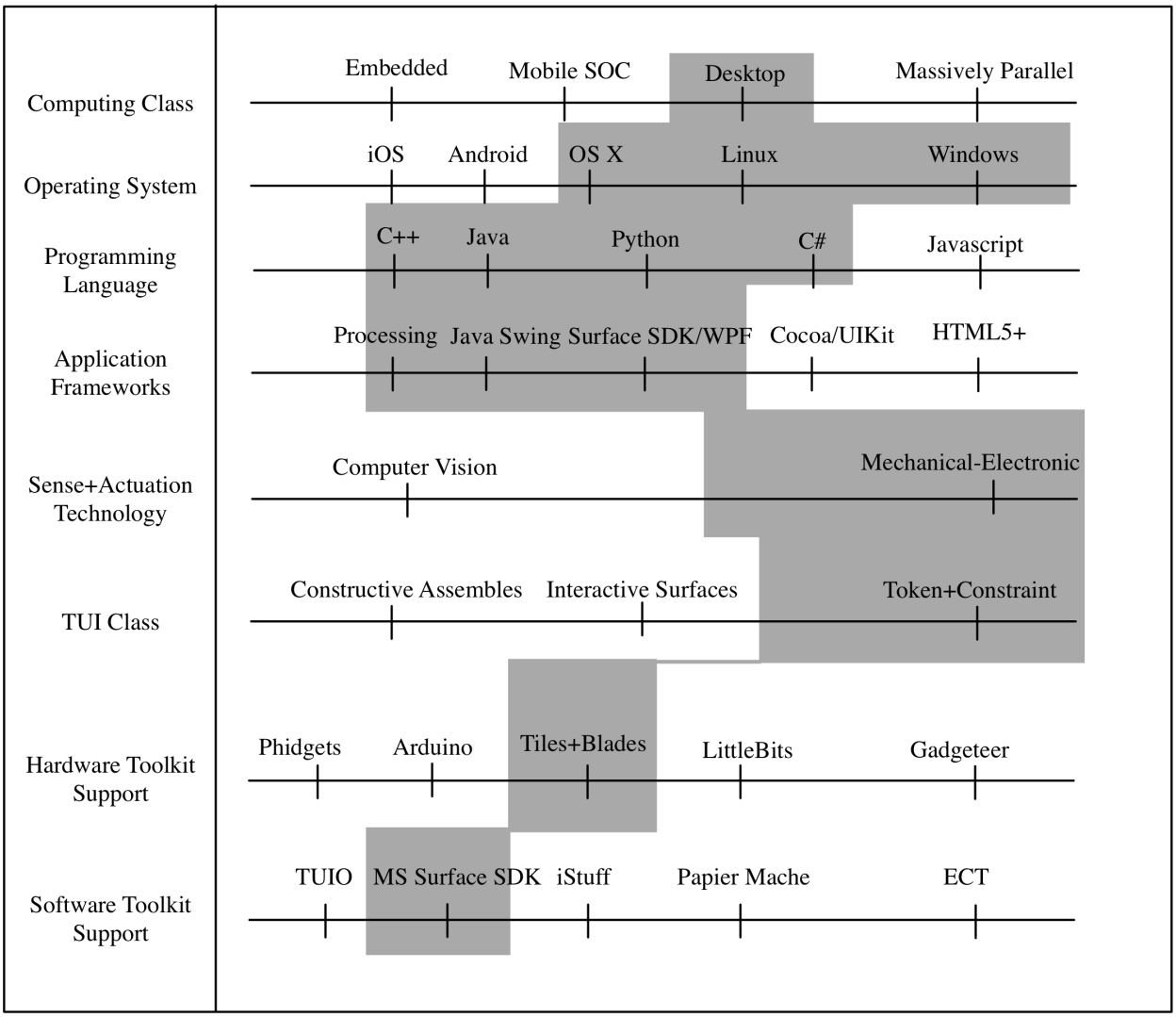

FIGURE 1.2: Dimensions of heterogeneity within the design space of tangible interfaces. The gray region represents areas within the design space that have been addressed by this dissertation's research.

1.1 Research Problems and Motivations

Solutions for the design, development and deployment of tangible user interfaces (TUIs) within distributed, heterogeneous computing environments must address the challenges of distributed systems and TUI development often in the pursuit of conflicting design goals. Additionally, progress is impeded in the distributed, heterogeneous TUI space is progress by the non-separation of incidental and inherent complexity; strong coupling of components within a distributed TUI; and strong coupling betven application-level code and the configuration of resources that constitute distributed TUI-based application.

As mentioned above, tangible interaction is a multidisciplinary endeavor, with each class of stakeholder facing challenges. To some degree this research presented primarily here addresses the concerns of 1. TUI designers/developers; 2. and to a lesser extent,TUI tool makers; 3. and infrastructure developers for distributed, heterogeneous TUIs. One common theme the list of stakeholder issues is the lack of a clearly defined requirements and specifications for the aspect of distributed tangibles-based system a stakeholder might be interested. Another theme is the lack of expertise or affordability of focus in addressing the complexity of infrastructure that supports a stakeholder's area of concern or expertise. (e.g. a distributed interface developer may not be expert in network programming). This speaks to the coincidental complexity of developing distributed tangibles-based systems. Another theme in the list of stakeholder problems centers around the need for distributed tangibles-based system flexibility to address issues such as the abundance of choice in technologies that can be used to realize a distributed TUI. For instance, both TUIs and distributed interaction face a set of open research problems; and so enduring tangibles-based applications require the ability to sustain changes in the underlying infrastructure. Problems such as these are the focus of this research, which has been pursued through the study of software architecture for TUIs with distributed, heterogeneous computing resources.

As will be elaborated in further detail later in this document, the functional requirements a system for include support for the integration of distributed, heterogeneous resources for tangible interaction and management of the life cycle of such resources. As a developer of tools for designing and building TUIs, one must consider decisions about the dimensions of tangible systems distributed and heterogeneity such as sensing+actuation technology (e.g. visual tracking versus mechatronics), end-user application platforms (e.g. Processing, HTML5, etc.), and communication protocols to integrate remote resources (e.g. TCP sock-

ets, XMLRPC, TUI/OSC, etc.). See Figures 1.1 and 1.2 for illustrations of the design space of distributed, heterogeneous systems for tangible interface-based applications.

This research is motivated by experiences in developing tangibles to support the real-time interactive scientific visualization and related computational science tasks. What is interesting, from an interactive systems perspective, about these use contexts, is that they typically involve computational environments that consists of distributed computers for which its users are interested in fine-grained manipulative control of elements within a system within a visualization that may be generated by a range of system hierarchies (e.g. a single supercomputer class system provisioned for large data visualization, a collection workstations and laptops running individual instances of a visualization system for small scale datasets, etc.). The author has been involved in several efforts to realize systems that utilize the cutting edge technologies for high speed networking (gigabyte data throughputs over global optical networks), parallel computing (thousands to hundreds of thousand cores), data visualization (datasets weighing hundreds of gigabytes to terabytes) and user interface (remote, distributed tangible interaction) to facilitate scientific collaboration across physical, organization and disciplinary boundaries [HAB+06, HAG+09, UHBH03, UJRT07]. This dissertation's research is motivated by a future in which such activities are commonplace and thus seeks to explore architectures and tools that address the complexities posed by the design, development and deployment of TUIs within distributed, heterogeneous computing environments.

More specifically, if we consider the use case of remote distributed interactive visualization as supported by tangibles, such a must address the following issues:

- support the acquisition, transport and handling of user events at a rate 10/second or 100ms per event for real time interactivity and at least 1 event per second for supporting the perception of causality.
- for systems involving multiple devices and/or users, support the event throughputs as the number of nodes increase
- support extensibility in the type of implementations of tangible interfaces and visualization systems

But given these requirements, we must also consider the targeted users, namely user interface developers and computational science experts. Neither class of user are expected to be expert in distributed interaction programming or software/hardware engineering for tangible user interfaces. As such, low thresholds and and high ceilings are desired properties for tools and environments for building tangibles in distributed, heterogeneous computing environments [MHP00]. To lower the threshold of building distributed tangibles,

a system architecture might be designed to reduce the incidental complexity of network programming; while high ceilings might be realized by providing computational support tools to guide the path from high-level specification to the range of final configurations that can be realized with the tools.

Thus far, discussion has been given to the technical challenges to be addressed by tools for building distributed, heterogeneous tangibles. But what of the challenges to designing, building and refining the tools themselves? Many of problems of distributed interaction, and developments tools as user interface are beyond the scope of single dissertation project or organization. The distributed interactive systems ecosystem envisioned by the author and his collaborators consists of increasingly numerous and diverse resources. And so any tool that seeks to address a significant subset of this ecosystem, must provide an architecture that can be adapted to this ecosystem. The field of software architecture can address the issues of scope by structuring both the requirements and the systems that seek to support those requirements. Software architecture facilitates communication among stakeholders. And based on Fielding's definition of software architecture, it can facilitate thinking about a system's operation regardless that system's state of implementation [Fie00].

The author set out to address the above-described problems of distributed, heterogeneous systems for TUI-based software initially through development of a software development toolkit. After several iterations on the basic functionality of this toolkit, the author began a study of the software architectural principles that would both inform further development as well as evaluation of any software engineering related to such tools for distributed TUIs. For instance, if one distributed, heterogeneous TUI toolkit design goal was to support interactive responsiveness that scales in the number devices that comprise the system, a study of software architecture could illuminate system design that support or constrain scalability. Additionally, in the pursuit of scalability, software architectural thinking could help identify what other properties might be traded-off.

1.1.1 Architectural Support for Tangible User Interfaces with Distributed, Heterogeneous Computing Resources

In this dissertation, an architectural specification is introduced to address the challenges of designing and developing distributed tangible interfaces. This architecture is based on abstractions to separate the specification of a TUI's structure and behavior from its configuration as loosely coupled, message-passing components within a distributed, heterogeneous computing environment. From one perspective, this architecture

consists of three basic layers: a resource layer, a message-based communication layer and a layer consisting of application-level components. At the resource layer of the architecture are modules that act as adapters for distributed TUI resources (e.g. sensors, actuators, tangible devices and input and display, applications and services, etc.). The interface of each adapter is characterized by the set signals or events it produces and the set of stimuli or signals it responds to. At the application layer are a collection of resource proxies that expose an application programming interface (API) for integrating distributed TUI resources within an application. Each class of resource proxy implements the same interface as the resource adapter to which is bound during runtime. The resource and proxy layers communicate via a messaging layer, which abstracts details about the physical channels over which components communicate. The structure and behavior of a tangibles-based application is defined within this architecture by binding each proxy within that application's context to one or more resources via their adapter interface and specifying event-driven behavior. The ability to bind and rebind a collection of proxies to a variable collection of resources via their adapter enables applications based upon this architecture to vary in the number, type and location of resources that employed by an instance of distributed tangibles-based application.

This architecture is named the Proxy Tangible Interactor (PTI). The term *proxy* is used to denote the fact that application-level definition of consists of abstract representations of concrete TUI elements which may exist anywhere within the distributed computing environment. The term *tangible interactor* is used to denote the interface that is shared between a resource's proxy and adapter, as well as a nod the abstraction for tangible input integration on which this interface is based [KL09]. To further explore properties of this architecture, a toolkit consisting of a small class library for TUI development and lightweight communication middleware was developed and is presented here.

1.1.2 Thesis Contributions

This dissertation makes the following contributions to interactive systems software research in the field of computer science:

- The identification of PTI, a software architecture that supports tangible user interface-based applications that employ distributed, heterogeneous resources.
- A software development toolkit based on the PTI architecture, that enables tangibles-based applications to vary their configuration in the number, type, and location of resources.

1.2 Dissertation Overview

This dissertation is organized into six chapters. In Chapter 2, related literature is discussed. This chapter covers background literature on tangible user interfaces, distributed computing and software architecture. The chapter then turns its focus on frameworks and toolkits for realizing tangible user interfaces and related user interface approaches.

In Chapter 3, discussion is given to the topic software architecture for distributed tangible user interfaces. It begins with a section on requirements analysis for distributed tangibles. Based upon these requirements, an architecture is specified through the use of architectural diagrams that illustrate the various architectural viewpoints with respect to stakeholders and phases of the system design and implementation phase. The chapter then goes on to describe the major elements of the PTI architecture for distributed, heterogeneous tangible interaction.

TUIKit, a PTI-based toolkit is presented in Chapter 4. This toolkit is discussed in terms of its architecture, implementation, related work, and strengths and limitations.

Chapter 5 and 6 provide discussion and conclusions of the research presented in the dissertation.

Chapter 2

Background

The proposed work builds on prior research in the areas of tangible user interfaces, distributed computing and software architecture. This chapter discuss relevant literature from these fields, how this work is related and how it differs. As described in the previous chapter, the main thesis contribution presented here is a software architecture for systems for tangible interaction in distributed, heterogeneous computing contexts. In §2.1, literature on tangible user interfaces is discussed in terms of the field's history and potential trajectory.

As mentioned in the previous chapter, computing trends imply a near future in which computation of interest to users is spread across space and time over diverse system resources. In light of this, this chapter examines the literature on distributed, heterogeneous computing to identify the advantages, challenges and strategies that would inform the architecture of tangible interaction systems in such computing contexts (see §2.2).

Software architecture is concerned with structure of software based systems. The processes of designing, developing, refining and deploying tangible user interfaces is a complex process involving several areas of expertise and many potential stakeholders. The field of software architecture offers tools to organize the needs of stakeholders and the structures of elements within target systems [GAACB95]. In §2.3, the chapter provides discussion of the software architecture.

In the study of tangible user interfaces, perhaps the area most closely related to software architecture is the area of frameworks and toolkits for tangible interaction. It is in this literature, we are most likely to see the description of an architecture for the range of system that can be realized with a given toolkit or framework. In §2.4, discussion in given to the area of frameworks and toolkits for tangible interaction.

2.1 Tangible User Interfaces

Tangible user interfaces (TUIs) "give physical form to digital information, employing physical artifacts both as representations and controls for computational media [UI00]." Tangible interaction, as of this writing, is an active area research focused on linking the physical and digital worlds, whose vision was initially articulated in the late 1990's [IU97]. TUIs or tangibles date as far back as the 1970s with Perlman's SlotMachine

[Per76] and include several systems which predate the term “tangible user interface” [HPGK94, FIB95]. As evidence of the growth of the field of tangible interaction, today hundreds of papers are published yearly in a multi-disciplinary conference dedicated to tangible interaction and related research, in addition to mainstream research publication venues in human computer interaction. We also find several examples of tangibles as mass-market products or mass-market products that are influenced by tangible interaction-based approaches [rea12, sif12, spi12].

2.1.1 Conceptual Frameworks for Tangible Interaction

While the intellectual focus of TUIs research is often on the physical artifact, many TUI-based systems are software intensive. Thus is reasonable that progress of the field is driven by breakthroughs in software technologies as it is driven by progress in physical design and hardware technology. As such the insights of Redwine and Riddle on the maturation of software technology is relevant to the field of tangibles-based software and tangible interaction [RJR85]. Redwine and Riddle propose that software technologies goes through phases of maturity, which begin with basic research and conceptualization and culminate in popularization. Here we are concerned with the basic research and conceptualization phases of the tangibles research and development and so we find several in the literature who have contributed to conceptualization of tangibles [Dou04, Fis04, HB06, IU97, SWK+04, UI00].

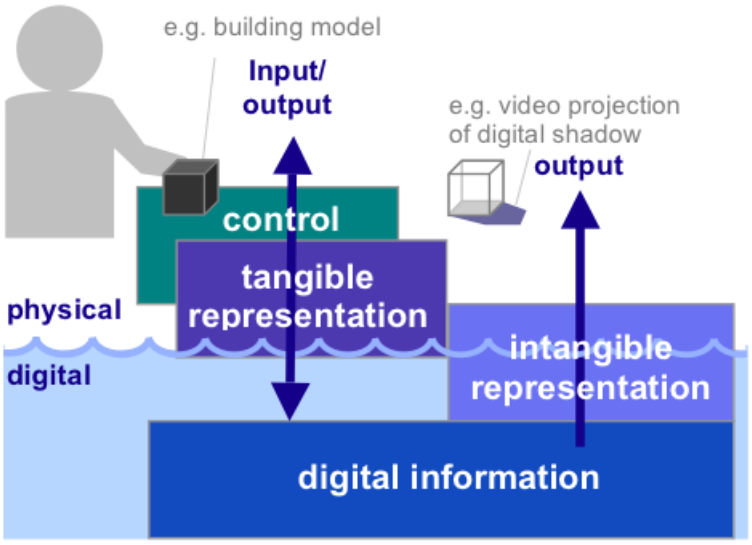

FIGURE 2.1: Tangible Bits: A Conceptual Diagram of the Components of a Tangible Interface [Ish08]

Tangible computing systems incorporate physical control and representation of digital information [Dou04]. According to Ullmer and Ishii, tangible user interfaces (TUIs) are characterized by

1. a computational coupling of physical representations to digital information;
2. an embodiment of control mechanisms by physical representations;
3. a perception of physical representations as being coupled to digital representations;
4. and the embodiment of key aspects of system's digital state as the tangibles' physical state [UI00].

MCRit Ullmer and Ishii introduce the Model–Control–Representation(intangible and tangible) (MCR_{it}) (see Fig 2.1), which is an extension of the Model–Control–View design pattern, as a conceptual framework for distinguishing tangible interfaces from other interaction approaches [UI00]. The original MVC pattern is an architectural pattern that decouples aspects of graphical user interfaces for purpose of increased flexibility. For instance, a GUI application's view and input from its underlying model enables one to use multiple representations of the same data. The model is an object that encapsulates application domain functionality and data. With tangibles, the view component is split into a tangible and intangible representation component. A tangible representation comprises of the physical elements of a TUI. Intangible representations includes graphical and aural output and endow TUIs with the flexibility to represent information which may dynamic or infeasible to represent tangibly. The control component provides mediation between the view and model within this composite.

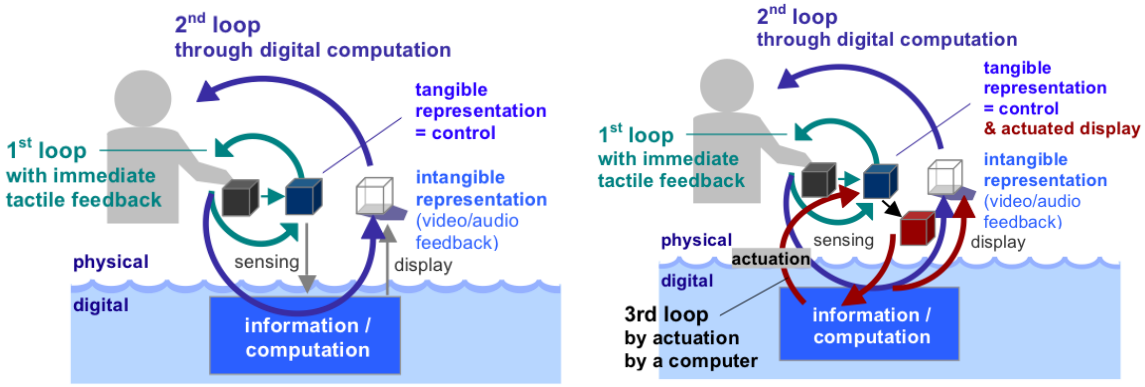

FIGURE 2.2: Dual Feedback Loops of Tangible User Interfaces [Ish08]

Taxonomy for Tangible Interfaces A taxonomy based on the two axes of metaphor and embodiment is proposed by Fishkin in an attempt another tool for analysis and to tease out design principles for tangible

interaction design [Fis04]. Fishkin introduces a broad definition of TUIs characterized by a processes in which an input event, typically a physical user manipulation, triggers the computer system. The computer system then senses the event to change some state and provide feedback as an output event, which causes a change in the physical state of some object. The classification of TUIs is based, not on a binary designation, but on a continuum of tangibility. Fishkin agrees with Ishii and Ullmer that this relaxation affords the consideration of interesting interaction techniques in the tangible category.

Heuristics for Spatial Tangible Interfaces Sharlin et al. present of set of observations of TUIs, and from these observations, introduce several heuristics for analysis and incorporation of spatiality within TUIs [SWK+04]. These heuristics include spatial mapping, I/O unification and support for ”trial and error.” This spatially-based heuristic provide yet another tool for comparing and contrasting TUIs and non-TUIs as well.

Getting a Grip on Tangible Interaction Hornecker and Burr describe tangible interaction as denoted by systems “rely on embodied interaction, tangible manipulation, physical representation of data, and embeddedness in real space.[HB06]” The authors present a framework for analysis of tangible interaction systems that examines the interplay of the material/physical and social aspects of the larger context in which the tangible interaction system exists. Hornecker and Burr also introduce several viewpoints of tangible interaction:

data centered view A perspective that focuses on the relationships between the physical embodiments of control and representation and digital information. It focuses on the tangible user interface

expressive-movement centered view This perspective seeks to design the interaction itself. This school focuses on tangible interaction.

space centered view This view focuses on the physical spaces in which devices for digital display and tangible interaction are embedded. It may also focus upon the body as interaction device and display.

The framework presented by Hornecker and Buur is composed of several themes: tangible manipulation, spatial interaction, embodied facilitation and expressive representation. Because this thesis is concerned with systems which support tangible interaction in distributed, heterogeneous computing environments, this research is grounded in the data-centric view. This work is also focused on computational tools that facilitate the tangible manipulation of computation to achieve some end of interest to the user.

2.1.2 Strengths, Limitations and Challenges of Tangible User Interfaces

Fitzmaurice et al, in their introduction of the concept of graspable user interfaces, which may be considered a subset of tangibles. Graspable interfaces focus on the manipulation of physical objects to achieve some digital end. Some strengths of graspables as presented by Fitzmaurice include:

- it can support two-handed interaction
- brings focus on context-sensitive input devices
- enables parallel user input
- leverages well-developed human skills for physical manipulations
- externalizes internal digital representations

It is my belief that tangibles inherit many of the strengths of graspables. The aim of this work is to research and develop software tools which aid developers and end-users in the specification and use of tangibles, specifically for distributed visualization applications.

Shaer and Hornecker in their survey of TUI literature, discuss the strengths and limitations [SH10]. TUIs strengths include:

Collaboration TUIs are often a good fit for collaborative use contexts. This has been demonstrated by several instances of TUIs designed for collaborative applications [Ais79, FFF80, SK95, Jor10, UI99]. Hornecker and Buur list as factors supporting co-located collaboration 1. familiarity and affordances of everyday physical interaction with the real world lower engagement threshold; 2. the inviting quality of tangibles; 3. and enhanced legibility due to the observability of manual interaction.

Situated-ness Tangibles are situated in the physical world, not simply on virtual displays [Dou04].

Tangible-Thinking TUIs support thinking to physical action or tangible thinking [KHT06]. Some forms of tangible thinking include gesture, distributed cognition and tangible representation.

Space-Multiplexing and Directness of Interaction The use of multiple physical artifacts within a TUI, enable each embodiment of control and representation to be engaged independently and simultaneously [Fit96]. This also enables tangible representations to have more specific forms as opposed to generic forms to support general purpose.

Strong-Specificity Enables Iconic Representations and Affordances More specific forms enables designers to use representations that are recognizable and communicate a tangible's affordances.

TUIs limitations include:

Scalability and Risk of Losing Physical Objects Several limitations of TUIs fall under the category of scalability [SH10]. Very few TUI applications scale up to large problems involving large data sets or many parameters. The use of physical artifacts within TUIs has implications for clutter of interactive surfaces, inflexibility (inability to scale or modify scope of representations), and difficulty of support complex commands because of a high degree of direct manipulation [BBE+02].

Malleability and Versatility The materials in TUIs are not as malleable as are elements of graphical user interfaces (GUIs). Physical objects are typically static and rigid [PNM+04]. This can limit the versatility of TUIs. Furthermore, TUIs are more limited in the range of tasks that can be support. Jacob et al. suggests that interface designers consider the tradeoffs of reality-based interfaces approaches such as TUIs between realism and versatility [JGH+08].

User Fatigue Because TUIs use physical forms, TUI designers have to consider issues ergonomics and long-term strain as it relates to manual actions required by TUIs [SH10].

Above the strengths and limitations of TUIs have been overviewed. In many cases the strengths of TUIs are linked to their weaknesses, and so TUI designers must be able to evaluate the tradeoffs of using a particular tangible interface element in their applications. In the course of considering strengths and weakness of TUIs when designing, developing, and deploying tangibles, TUI researchers and practitioners must also faces the open challenges of realizing tangibles.

Based on upon Redwine's stages of technology development [RJR85] and through examination of TUI research literature, it might said that TUI research and development are is primary phases of internal enhancement and exploration. Many techniques for tangible interaction have been published and disseminated throughout the community. We also find several instances of toolkits and frameworks for integration of tangibles within applications [GB02, KL09, BRSB03, VG07, KB07a]. But as observed by Shaer and Hornecker, each time a new technique is introduced, a TUI developer has to learn a new toolkit or software library and rewrite their application code [SH10]. Another challenge of TUI development is that often these techniques within a standard set of widgets often excluding other interaction styles. Additional TUI development challenges are outlined by Shaer et al. [SJ09]. As remarked by Streitz at an international conference on tangible interaction, for TUIs and related interface approaches to have more impact, they need to pursue larger scale applications [Str11]. This might involve applications involving TUI techniques and subsystems

from multiple organizations and target legacy applications and well novel applications. Such pursuits will involve diverse resources for computational, networking, and interactions, and so the aim of this research is approach the traditional challenges of TUI development in such contexts.

2.2 Distributed, Heterogeneous Computing

> The vast majority of all computing today is distributed. Local computing, which forms the roots of computer science and is itself still evolving, has become the exception rather than the rule [Gof03]

In the previous section, background literature on TUIs was discussed. In this section, background on distributed, heterogeneous computing is covered. A majority of the TUI-based systems discussed in the TUI literature at the time of this writing are either supported by local computing or do not emphasize the distributed nature of its computational back-end. Notable exceptions include systems by Brave and Ishii [BID98], Ullmer and Ishii [UI97], Jorda et al. [JKGB05], and Marquardt and Greenberg [MG07]. The intersection of tangible interaction and ubiquitous computing research has also produced several toolkits and frameworks for building distributed reality-based interfaces [BRSB03, Hol09, KB07b, KTK+05, MG07]. One premise of this research is that interactive computing will occur over distributed resources, which means that interactive system design will have to address the both requirements of interactive computing as well as distributed computing.

In pursuit of that effort, the question becomes, "How are local and distributed computing systems different?" Distributed systems are systems composed of nodes with different address spaces [KWWW94]. Waldo posit that all the differences between local and distributed computing originate from four differences: 1. latency, 2. memory, 3. concurrency, 4. and partial failure . The implications of these differences are summarized by eight fallacies of distributed computing that was elaborate by engineers at the former Sun Microsystems [DG97, Gof03, RGO06]:

1. The network is reliable
2. Latency is zero
3. Bandwidth is zero
4. The network is secure
5. Topology doesn't change

6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

With respect to interactive system design, the fundamental differences of distributed computing and its implications mean that the networking aspects of an interactive system cannot be simply masked [KWWW94]. Interactive systems in a distributed, heterogeneous computing context should be designed to mitigate the complexity posed by such contexts.

Now that the differences between local and distributed have been outlined, let us consider the benefits of distributed computing as enabled by the network as mediator The fundamental benefit of distributed computing is that through networks people can access four types remote resources [Dix08]:

people Networks can facilitate communication with remote people.

physical things Through networks, we can view and control remote physical objects.

data Networks mediate remote access to data.

computation Networks can grant access to remote computational resources.

Given these benefits of networks and distributed computing, in the following subsections several distributed computing technologies are discussed.

2.2.1 The Grid View of Distributed Computing

This author's first exposure to the field of distributed computing as research was through grid computing and "e–everything." Foster defines the Grid

> ... as an extensible set of Grid services that may be aggregated in various ways to meet the needs of VOs, which themselves can be defined in part by the services that they operate and share. [FKNT02]

The vision of a set of technologies and policies as infrastructure that would support the development of computing applications that span technical, geographical and organizational boundaries is an intriguing one. The *Grid* was proposed to present a unified system architecture and programming model to address the unique challenges of distributed, heterogeneous computing. Foster in his discussion of alternative perspective of grids [FKT01], claims that grid-specific programming models are not required for a grid architecture, although "abstractions and encapsulation can reduce complexity and improve reliability." The layered grid

architecture that I believe applies to other distributed computing paradigms, that organizes components and services into 1. fabric, 2. connectivity, 3. resource, 4. collective, 5. and application.

See Figure 2.3 for an illustration of the relationships between a grid architecture, an architecture for distributed tangible interaction and the Internet protocol architecture.

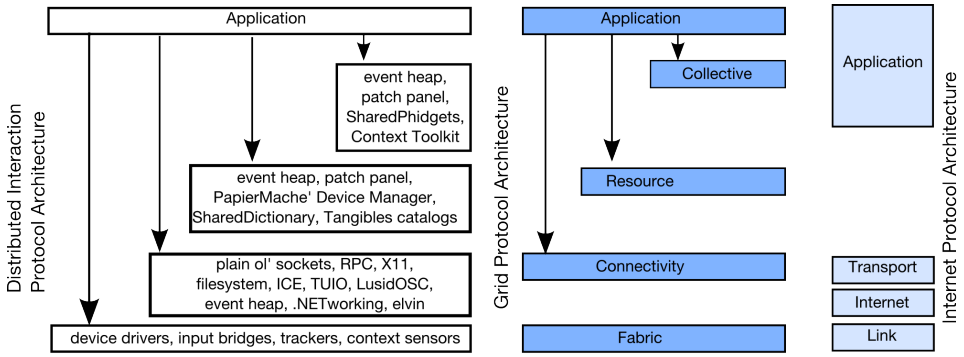

FIGURE 2.3: The relationship between the Internet, Grid and Distributed Interaction Architectures

Grid computing is not the definitive view of distributed computing, but in that paradigm we find a comprehensive view that considers many of the concerns of coordinating and composing applications from distributed, heterogeneous resources. In the future, it likely valuable to examine other distributed, heterogeneous computing paradigms and their approaches to software infrastructure (e.g. ubiquitous computing). It is possible that many distributed, heterogeneous computing paradigms are structurally similar, but vary fundamentally in the behavior of the components and services used in the composition of distributed systems.

2.2.2 The Ubiquitous Computing View of Distributed Heterogeneous Computing

Ubiquitous computing (ubicomp) is one paradigm of heterogeneous, distributed computing that is alternative to grid computing. Mark Weiser introduces ubiquitous computing as a vision in which computing recedes into the background, becoming seamlessly integrated into human beings' surrounding [Wei91]. This future is envisioned as having a plentitude of devices with various form factors and capabilities that combine to augment human intellect. The tangible computing goal of integrating physical interaction into computing systems is closely aligned with the goal of realizing calmer technology [WB96]. Currently, the author has not identified a architectural description of ubicomp along the lines of Foster's anatomy and physiology of the *Grid* [FKT01, FKNT02]; however, Weiser and later Kindberg and Fox address computer science and system engineering respectively of ubicomp [Wei93, KF02]. In the overlap between research

TUIs and ubicomp, a significant body of work address issues of distributed, heterogeneous computing of post-WIMP approaches to interaction such as TUIs. Such work includes systems such as iROS [JFW02], iStuff [BRSB03], the Equator Component Toolkit [GIM+04] and Concerto Widgets [KTK+05].

2.2.3 APIs for Distributed Application Development

The ideas represented by the Grid Application Toolkit (GAT) and Simple APIs for Grid Applications (SAGA) projects have greatly influenced this research toward simplifying distributed TUI-based application development [ADG+05, GJK+06]. The distributed computing community offer several protocols, services and components for realizing distributed computing systems. application programming interfaces (APIs) and software development kits (SDKs) like SAGA aim to provide generic interfaces to core distributed computing protocols and services that may be provided by a number of underlying implementation technologies. Similar approaches to tangible interface programming are embodied in toolkits like Papier Mache [KL09] and SharedPhidgets [MG07] interactive resource abstraction. The adaptor and proxy design patterns [GHJV95] are employed within TUIKit to provide for both fan-in (fusing input from diverse interactive resources) and fan-out (fissioning output to diverse devices and software components).

The work proposed here also seeks the leverage the knowledge contributed by the distributed and ubiquitous computing communities. The general idea is that distributed applications can become easier to develop, more reliable and take advantage of more code re-use through generic interfaces to services and resources despite differences in underlying technology. Efforts such as the grid application toolkit (GAT) and simple APIs for grid applications (SAGA) illustrate such a possibility.

2.2.4 Distributed Interaction: Discussion

Goff describes the space of networked-distributed computing (NDC) as a fitscape [1] and goes on to discuss several technologies and paradigms for addressing many of the challenges of NDC [Gof03]. Within respect to building interactive, ubiquitous computing environments Johanson and Fox provide a good analysis to how many NDC concerns are handled by various approaches to NDC infrastructure [JF04]. See Dix's survey on network-based interaction for additional analysis the challenges introduced to human computing interaction by distributed computing, as well some best practices for addressing them [Dix08].

[1] fitscape: "fitness landscape," an ecosystem in which technologies compete for adoption

In this section, the *Eight Fallacies of Distributed Computing* is used as framing mechanisms to consider the problems of distributed computing that must be addressed by software architectures that support tangible interaction in these contexts. Using this list (see §2.2) as a high-level view of the concerns to be addressed in distributed, heterogenous computing, the research presented in this document addresses concerns (5), (7), (8), and to lesser degree (1) and (2) in the context of tangible-based application development within distributed computing environments. In the next section, this chapter turns to the topic of software architecture. It is at the level of architectural design the above mentioned distributed, heterogeneous computing concerns are addressed.

2.3 Software Architecture

There is an ongoing debate on a consensus definition of software architecture that is agreed among the software architecture research and practice community [KOS06]. An ANSI/IEEE standards committee define architecture as

> the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution [IEE00].

Bass et al. define software architecture as

> is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [BCK03].

Abowd focus on the descriptive aspects of software architecture and define it as

> an important level of description for software systems. At this level of abstraction key design issues include gross-level decompositional components, protocols of interaction between those components, global system properties (such as throughout and latency), and life-cycle issues (such as maintainability, extent of reuse, and platform independence) [AAG95].

And finally, Fielding posits that software architecture need not be limited to just a description of a software system's structure, but

> an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture [Fie00].

Fielding's recursive definition is intriguing because it affords one the ability to use running systems as architecture; but more importantly its emphasis on abstraction gets to one of the core activities of software architecture, which is to simply. To systematically abstract details of a system's runtime enables one to reason about some aspect of a system while ignoring the details of other aspect irrelevant to the task at hand. Abowd's definition is also instructive on the what constitutes a software architectures and what information it should convey, but what if the description is executable? Perhaps given the many classes of stakeholders who may benefit from the process and products of software architecture, multiple definitions are useful to capture the concerns and viewpoints of these stakeholders.

2.3.1 The Elements of Software Architecture

As mentioned in several software architecture definitions, an architecture is defined in terms of the relationships between its elements. Perry and Wolf defined this as {elements, forms, rationale} = software architecture [PW92], extended by Barry Boehm as {elements, forms, rationale, constraints} = software architecture. The major architectural elements are *components*, *connectors*, and *data*. A *component* is a unit of processing that transforms data elements. *Data* is information that is sent to or received by some component via a connector. And a *connector* mediates communication, coordination, or cooperation among components [SC97]. Diagrams with boxes and arrows are common media use to convey architectures and can be very effective in communicating properties of an architecture. *Architectural constraints* restrict the structure and behavior of an architecture's elements.

With the basic elements of an architecture defined, discussion is now given to the aim of software architecture, which is to produce or derive an architecture with a certain set of properties. Architectural properties are those characteristics that derive from the selection, arrangement and arrangements of architectural elements within a system [Fie00]. Such properties include functional properties (i.e. what a system does) as well as non-functional properties, also known as quality attributes (e.g. evolvability, component reusability, performance, etc.). To achieve a set of architectural properties, constraints are applied to an architecture. For instance, to achieve a extensible or modifiable architecture, one might constrain components to communicate via event-based invocation[GS94].

Another important tool in the software architecture toolbox, is the abstraction of architectural styles. Fielding defines an architectural style as a coordinated set of architectural constraints that restricts the

roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style [Fie00]. Architectural styles enable one to consider (sub)system architectures at a higher level of abstraction and supports the comparison and evaluation of various architectures. This becomes useful because the knowledge in accumulated through practice and research about the properties belonging to a particular style, making it easier to reason about the properties that might be achieved as a result of a particular architectural design decision. For instance, to say that reacTIVision's architectural (cf. [KB07a]) incorporates a client-server hierarchical style is to say that reacTIVision's architectural is evolvable as client-server architectures are evolvable because of the separation of concerns design in the partition of work between client and server components [And91].

2.3.2 The Views of Software Architecture

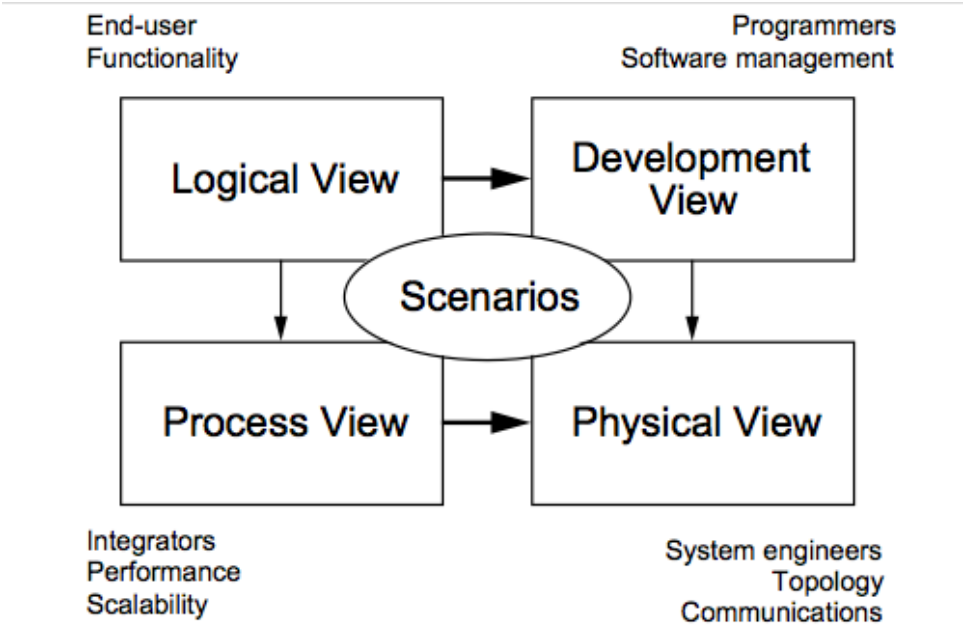

FIGURE 2.4: Krutchen's 4+1 View Model of Software Architecture [Kru95]

Above fundamental concepts of software architecture have been discussed. We now consider methods for describing and disseminating software architectures. A common practice of architectural diagramming, typically using boxes and arrows, is heavily influenced by Krutchen's paper on the *4+1 View Model* [Kru95]. This model is based on the premise that architectural design and evaluation is a complex process involving many parts and types of stakeholders and so it makes sense to view an architecture from different perspec-

tives. In Figure 2.4 the *4+1 View Model* is illustrated. Here the relationships between system architecture stakeholders and particular views can be seen. These views are as follows:

Logical Architecture View The logical architecture supports the functional requirements. This architecture is derived from a decomposition of a system into a set of key abstractions taken from the target domain. These abstractions are often expressed as classes or objects, thus enabling common mechanisms and design elements to be convey in terms of logical relationships (e.g. composition, inheritance, etc.) [Kru95]. In this research, this type of object-oriented notation is used, however Krutchen states that other domains may require alternative logical notions (e.g. a data-driven system may use entity-relation (E-R) diagrams (cf. [Che83]).

Process Architecture View A view of the process architecture enables one to observe the non-functional properties of a system. The process architecture addresses concerns of concurrency and distribution, integrity, fault tolerance and the partitioning of abstractions from logical architecture within the process architecture. A process architecture is defined in terms of a logical network of communicating processes, a process being a group of tasks executing on an object.

Development Architecture View The development architecture focuses on how software modules are organized within the software development environment. This architecture is typically modeled by diagrams of modules and subsystems diagram. Although the full development architecture cannot be specified until the software module organization is defined, basic rules for system organization can be made. This architecture is of particular value in to the development planning process.

Physical Architecture View The physical architecture addresses non-functional architectural properties such as availability, fault-tolerance, scalability, availability and performance. A physical architecture describes the networks of nodes over which software is executed.

Scenarios Scenarios, more general instances of use cases, pull the four architecture views together.

2.4 Toolkits and Frameworks for Developing Tangible Interfaces

In the previous section, the topic of software architecture was overviewed. One area of tangible user interface research where intersects with software architecture is the topic of frameworks and toolkits for designing, developing and deploying TUIs. Mazalek and Hoven present a meta-framework for classifying tangible interaction frameworks and toolkits by type and the facets of tangible interaction each framework addresses

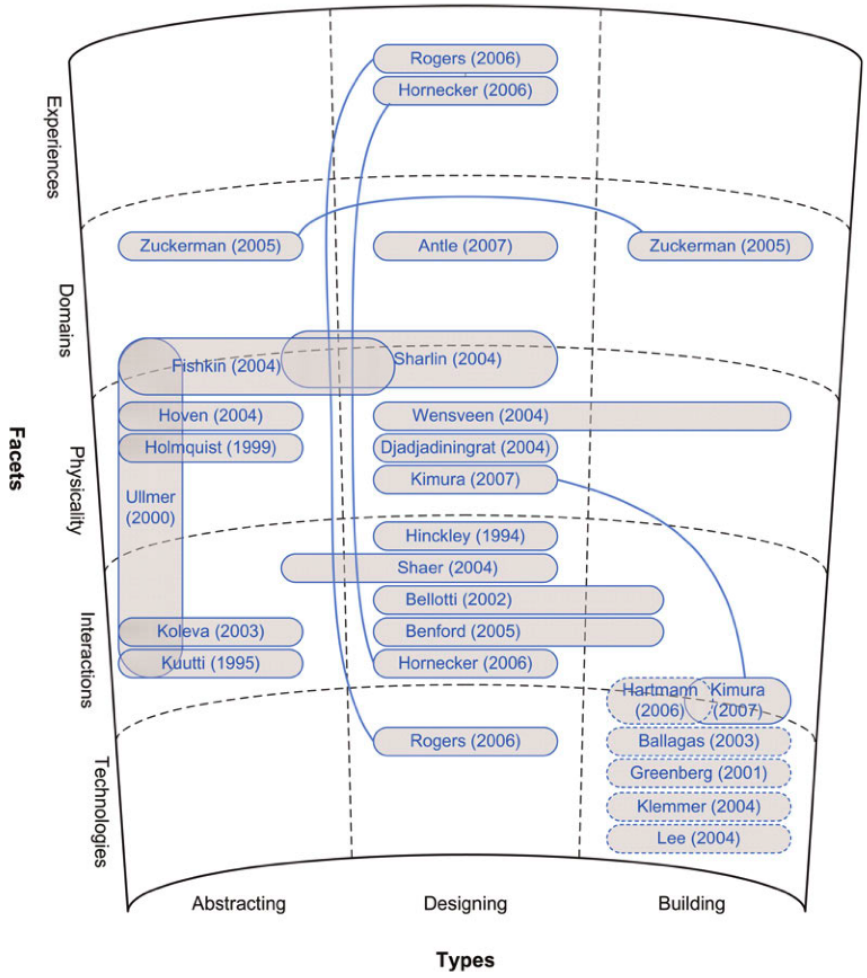

FIGURE 2.5: Mazalek and Hoven's map of tangible interaction and related frameworks [MvdH09]

[MvdH09]. As can be seen in Mazalek and Hoven's map of TUI frameworks (see Figure 2.5), these frameworks can classified by the phase in the TUI realization process they focus on, namely abstracting, designing or building. This research focuses on architectures related to building tangibles. This meta-framework becomes a valuable tool for tangible interaction designers and developers in deciding which frameworks to use. The framework also provides a map of the coverage gaps by existing tangible interaction frameworks. For instance, Mazalek and Hoven note very little attention is given by existing frameworks to specific domains. The architecture and as well as resultant toolkit and applications have mostly been in the domain of scientific visualization. A future research opportunity exists in applying to application components and services for data management and visualization those architectural principles (overviewed in §1.1.2 and expanded in §3.3) that have been applied to resources for tangible interaction in the research presented in document. In this section, discussion will focus on frameworks for the building and integrating TUIs. These frameworks are usually called toolkits and either focus on hardware components (§2.4.1) or software (§2.4.2). In §2.2, toolkits for developing distributed TUIs are discussed.

2.4.1 Hardware Toolkits

2.4.1.1 Phidgets

Greenberg and Fitchett present a toolkit for building physical interfaces with physical widgets or Phidgets [GF01]. Analogous to graphical widgets, phidgets abstract input or output devices and encapsulate higher level interactive functionality to the user. Phidgets are exposed to a programmer as ActiveX objects. These objects connect to a connection manager to keep account of device status. Greenberg and Boyle go on to describe additions to the Phidgets toolkit, which allow for Phidgets to interoperate with existing graphical applications [GB02].

2.4.1.2 LittleBits

LittleBits is an open source library of discrete electronic components aimed at democratizing the creation of electronic devices, especially for designers and artists [Bde09]. A goal of the project is to enable device makers to explore electronic functionality earlier in the design process. It achieves this by lowering the threshold of electronic fabrication through the use of magnetic connectors.

2.4.1.3 Tiles&Blades

Sankaran et al propose a tangibles hardware toolkit which aims to decouple the implementation of novel interaction hardware from the need to implement all of the required interaction modalities [SUR+09]. With the Tiles & Blades toolkit, interaction developers choose from a library of functional modules providing common modalities such as RFID sensing, physical controls such as knobs and buttons, and output such as LED and active haptic feedback. These modules are composed on Tiles, a physical substrate. Blades & Tiles are situated as a interaction hardware solution somewhere between custom electronics and fully-integrated, commercial off-the-shelf solutions. This toolkit provides developers the ability to extend existing hardware capability by designing new blades, which can be integrated with existing modules.

2.4.2 Tangible Computing Software Toolkits

2.4.2.1 iStuff

Ballagas et al. explore the construction ubiquitous computing environments with iStuff, a toolkit that enables dynamic mapping of devices to applications [BRSB03]. Applications constructed with iStuff are composed of devices that communicate over an intermediary communication channel called EventHeap [JF02]. This decouples device and application components, and with the addition of a component for wiring of input and output, PatchPanel, make iStuff applications highly modular and configurable. Applications built with iStuff are often distributed, though it has typically used for system distributed with a room. It is technically possible to use iStuff for remote distributed interaction, but certain usability issues have yet to be worked out. The decoupled inter-device communication at the core of this toolkit is an inspiration for the initial distributed interface component architecture that presented here. Such an approach enables interface components to be developed independent with a reasonable expectation of flexibility in composition and high re-configurability.

2.4.2.2 Papier Mache'

Inspired by a desire to integrate digital information with users' physical environment, especially paper-based interaction, Klemmer et al introduce the Papier-Mache toolkit [KLLL04]. Papier-mache abstracts input devices based upon computer vision and electronic-tagged (RFID) object recognition for integration in the development of tangible applications. This toolkit makes computer vision tangible application de-

velopment easier by enabling users to associate physical object manipulation while across multiple input sensor technologies and saving the user the trouble of managing low-level hardware communication.

There is currently no support in Papier-Mache for mechatronic-based tangibles like those built with Phidgets or Blades & Tiles. Papier-Mache also only currently supports input. For these reasons, I cannot use the toolkit as the basis for tangible integration. But this work, is exemplary for several reasons. Klemmer and Landay describe extensive work in the evaluation of this toolkit as an interface for application developers [KL09]. From examples included in their 2009 publication, it seems fairly straightforward to program tangible input. Papier-mache adapts Myers's Interactors abstraction for tangible input [Mye90]. This interaction model is being considered for extension to support output in the architecture and toolkit at the center of this research.

2.4.2.3 reacTIVison/TUIO

ReacTIVision is a toolkit for computer-vision tabletop tangible applications [KB07a]. Applications built with earlier versions of the reacTIVision toolkit consist of physical objects, which are tagged with fiducial markers, which are tracked via a object tracker which analyzes video streams to recognize symbols within the fiducial tags. Later versions of the toolkit also recognize blob objects for multi-touch tabletop applications. The toolkit consists of a recognition component, a communication component and a client library and has been ported across several operating system and application development platforms. The core communication protocol, TUIO, is a Open Sound Control (OSC) based message format, in which the presence and orientation of all recognized objects are constantly transmitted over a UDP protocol [KBBC05, WF97]. The reacTable, perhaps the most widely known tangible application built with on the reacTIVision platform, is real-time music application in which physical objects are manipulated to change various associated sound parameters [JKGB05]. User, multiple at the time even, can use reacTable as a musical instrument.

The reacTIVision system is attractive for building spatial tangible applications because it supports tracking of fingers and objects tagged with fiducial markers. The TUIO protocol is simple, but powerful and is being considered as the low-level protocol for communicating interaction data between components within the proposed toolkit. Using the TUIO protocol would open the possibility to integrate with many for which TUIO support has been implemented [tui11].

2.4.2.4 Shared Phidgets

To support the construction interfaces comprising of distributed phidgets, Marquardt and Greenberg present Shared phidgets [MG07]. Shared phidgets is built around the model of dMVC, an extension of the MVC design patter typically used for GUI applications. The MVC is extended for distributed interfaces by using a intermediary communication mechanism called the SharedDictionary. Phidgets are connected to a host computer, typically via USB, and are added to the SharedDictionary. The user may also add metadata on the type of device or geospatial information making that device searchable via queries to the SharedDictionary and discoverable. SharedPhidgets makes it easier to programming distributed physical UIs by providing developers several abstraction level at which to programming tangible interaction. Users may program physical UIs by directly accessing the SharedDictionary, or via a library of .NET objects which encapsulate Phidgets connected to the SharedDictionary, or by composing interfaces using interface skins that provide a graphical representation of phidgets.

2.5 Discussion

This work is closely related to the research literature on tangible and post-desktop interaction toolkits and application of tangible interaction to data visualization. I have overviewed much of this literature earlier in this chapter, but here I discuss several points of distinction between these works and the work I propose here.

The proposed work is distinguished from the Papier Mache' in that we support mechatronic tangible input and integration of distributed tangible interaction components. While SharedPhidgets was designed for building distributed information appliances from physical user interface elements, this work is targeted at the support fine-grained interactive physical control of and tight perceptual coupling to computational objects. It is quite possible that the SharedDictionary on which SharedPhidgets is built could be adapted or extending for our purposes. Because integration of networked resources for interaction are exposed in a network-transparent manner, it should be possible for integrate tangible interaction components via the SharedPhidgets infrastructure.

The proposed work is very much inspired by ubiquitous computing toolkit research involving eventHeap, iROS and subsequently iStuff. The incremental innovation from spatial and temporal decoupling as enabled by eventHeap to semantic decoupling via PatchPanel enables spontaneous integration of interactive

resources in very fluid manners. Related to this line of work, Johanson and Fox provide excellent analysis of the communication infrastructure requirements for ubiquitous computing systems [JF04] . Because the proposed toolkit is intended to support interactive response times and potentially scale to many users and interaction device instances, I chose to base my coordination channel on infrastructure aimed at support $\sim$ 1000s of message per second throughput [zer10].

The tangible interface-based applications described in this proposal and in prior work are primarily aimed at supporting scientific visualization tasks. In the literature we find several applications of tangible interaction techniques for visualization [BAC$^+$08, HPGK94, KKVL05, QM05, QMvLK05, RWP$^+$04, Sub04] . Most of these interfaces incorporate tangible interaction techniques for spatial manipulation of visualization assets. The proposed work toward visualization tangible applications is differentiated in that is possible to provides physical means of engaging abstract, non-spatial aspects of visualization. The proposed work is also differentiated by the scale and scope of system resources with which tangibles can be deployed.

Chapter 3

An Architecture for Tangible Interaction-based Systems with Distributed Heterogenous Resources

Several computing trends imply that computation of interest to users will become increasingly distributed over heterogeneous resources. The aim of this research is to identify architectural issues in the design, implementation and operation of tangible interaction-based applications in distributed, heterogeneous computing contexts. Engineering network-based applications in general is challenging for a numbers of reasons that are summarized by Deutsch et al.'s Eight Fallacies of Distributed Computing [Deu01]. With respect to interactive systems, Dix highlights the implications of these incorporating networks with interactive systems and argues for the need for coping strategies, algorithms and software architectures to mitigate the challenges of integrating user interfaces and network-based resources [Dix08]. This work builds upon a considerable body of research and development toward making it easier to build tangible and related reality-based interactive (RBI) applications [JGH+08]; however, this dissertation posits that efforts toward realizing tangibles for distributed, heterogeneous computing environments can benefit from a software architecture-level approach. Based on this premise, a goal of this research is to provide an architectural framework that provides support for abstractions to mitigate the complexities in realizing tangible user interface (TUI)-based applications within distributed, heterogeneous computing environments Some challenges of tangibles-based application design and development include but are not limited to:

- Supporting integration with diverse tangible input and output devices (e.g. mechatronic verse computer vision; toolkits/frameworks for interface software and hardware; diverse classes of computational device ranging from embedded systems to desktop to massively parallel systems; and diverse software stacks in terms of operating systems, programming language and application frameworks);
- Providing extensibility for integrating new types of devices, applications, and services as well as legacy systems [GB02];
- Providing abstractions that incorporate models of tangible interaction resonant with particular domains. For the Papier Mache toolkit supports basic media playback for object-based tangibles; whereas

several tangibles support software visualization actions throught manipulation of physical tokens and tools [HPGK94, MQA+04, CRR08].

As discussed in the previous chapter, the tangible and embedded interaction research community has made significant progress in making it easier to design and prototype tangible interaction system [BRSB03, GF01, KBBC05, KL09]. Despite conceptual and technical progress, many research questions remain open regarding the design and development of large-scale tangible interactive systems.

Such question include:

- What are the general concerns for distributed tangible interactive systems found in the TUI literature [BRSB03, Hol09, JFW02, MG07]?
- How can these concerns be addressed in distributed, heterogeneous computing contexts? For instance is it possible to provide the same API for integrating tangibles that supported by several distributed computer infrastructures? Alternately, how might a TUI developer in the design phase of a distributed tangibles-based application be shielded from the coincidental complexity of network programming?
- As a variation on the previous question, what abstractions are common across tangible interaction system and how might these abstractions be extended to distributed, heterogenous computing systems?
- What are the strengths and weaknesses of the various approaches to distributed, heterogeneous TUI-based applications? For instance, TUIs supported by infrastructure such as ubiquitous computing middleware (e.g. iStuff, Equator Component Toolkit, Concerto Widgets [BRSB03, GIM+04, KTK+05], etc.) support relatively easy composition of distributed interactive systems, but have tradeoffs in scalability in the number system nodes it can support.
- How can one design architectures to support system properties not previously achievable given certain constraints? Is it possible to support applications with millions or tens of millions of simultaneous, real-time interactive elements; what type of architectures would support such scale?

These are but a few of the architectural questions to be considered but we can begin by considering the system requirements of distributed tangible interaction. In the next section both functional and non-functional requirements for tangible interaction with heterogeneous resources are discussed. In §3.2, an architecture for tangible interaction with distributed, heterogeneous resources is described from several viewpoints. And based on the information revealed in §3.1 and 3.2, an architecture for distributed, heterogeneous interaction is describes in terms of its architectural elements (§3.3).

3.1 Requirements of a Distributed Tangible Interaction System

Our goal is to support the design, development and operation of tangible user interface(tangibles)-based applications for distributed computing contexts. Tangibles-based applications are those in which changes on physical objects result in changes within digital systems or vice versa. Physical objects can be used both as embodiments of digital information and well as handles to processes on digital information [IU97]. Underlying many implementations of such systems are diverse collections of sensors, actuators, computational and network communication devices. In the tangible interaction research literature we find many point designs and several tangibles-based system frameworks, often deployed on very small scales with limited interoperability between technical infrastructure. What we find lacking is an architectural description of a tangible systems that could support interaction with many physical objects across multiple locales and can effectively employ diverse implementation technologies. We are interested in applications that involve the fine-grained manipulation of tangibles that are linked to digital information that may be physically hosted remote from the point of user interaction. Conversely, a tangible may embody computation representing many data elements of varying dynamism (e.g. a ambient information appliance linked to seismic activity vs. one linked to presence of a single individual). Either possibility implies the desirability for communication infrastructure that can support sustained high throughput for network-based operations. To the extent that such applications would need to support multiple users at multiple sites, the infrastructure for high-throughput interactivity would need to be scalable as nodes are added to the system. This section discusses both the functional and non-functional requirements of systems that support tangible interaction through the use of distributed, heterogeneous resources.

3.1.1 Functional Requirements

Functional requirements define what operations for capability a system mush support. A tangible interaction-based system incorporates physical embodiments of digital information and processes. Such a system may involve the acquisition of input resulting from physical actions and invocation of a desired (or intended) software action. A tangible system may also actuate the physical world as a result of some state change within a digital system. A key property of tangible interaction user experience is the perception of linkages between physical world and computational system phenomena [UI00]. Additionally a tangibles-based sys-

tem may incorporate non-physical representations (e.g. audio or graphical displays) to present information pertaining to the artifacts in the tangible system or digital system.

Functionality for tangible system input and output is provided respectively by sensors and actuators, which may be embedded within a number of devices. For the purpose of tangible system integration, some entity (human or digital) must have knowledge of the availability, state and behavior of tangible interaction devices. *So one functional requirement of a tangible interaction system architecture is support for management of the the tangible interaction device life cycle*. This life cycle includes design, fabrication, connection to some computational system, disconnection from some system and interaction with elements within a computational system. To achieve perceptual linkages of physical representations and digital information, tangible interaction system resources must be integrated in a manner in which the semantics of those linkages can be defined. *Support for tangible interaction system resource integration is another key functional requirement of a tangible system architecture* Tangible system resource integration can be classified broadly into two types of processes: the transformation of physical action into software actions (tangible input) and the transformations of the digital state into physical action (tangible output). The tangible input process can be broken down into these stages:

1. acquire and disseminate input signals
2. interpret and filter input
3. invoke software actions
4. display feedback/feedthrough

The tangible output process involves the following stages:

1. access and disseminate digital state
2. interpret and filter digital information
3. actuate cyber-physical system
4. display feedback/feedthrough

Systems that fulfill the key tangible interaction system functional requirements mentioned previously must also meet additional requirements based upon the contexts in which the targeted tangible system is designed, implemented and deployed. Such contexts include properties such as:

- The locality of users, devices and computational resources;
- And technical characteristics of tangible system resources including:

- Sensing and actuation techniques;
- Software and hardware implementation platforms;
- and resource capabilities and affordances.

Such contextual system properties speak to the distribution and heterogeneity of resource that constitute a tangibles-based system. Tangibles-based systems of interest in this dissertation might involve users, interface elements and software instances which are distributed across networks of various physical scales (e.g. room-scale, building-scale, national-scale, intercontinental, etc.) and realized through diverse interaction hardware platforms; interaction techniques (mechanical-electronic vs. computer vision-based interaction techniques); and infrastructure for realizing tangibles (e.g. reacTIVison/TUIO, phidgets, arduino, etc.). With respect to distributed, heterogeneous computing contexts this dissertation identifies two functional requirements for such software architectures: 1. *Software architectures that support tangible interaction with distributed resources must provide support for the identification, connection and integration of remote resources across network connections;* 2. *Software architectures that support tangible user interfaces composed of heterogeneous resources must provide abstractions to integrate each type of diverse resource.* In §3.1.2, discussion is given to the non-functional properties of software architectures for distributed, heterogeneous tangible interfaces. These properties speak to the qualitative and quantitative degree a system supports the functional requirements described in this subsection.

3.1.2 Non-Functional Requirements: System Qualities

In the software engineering literature we find many definitions for non-functional requirements (NFRs). See Glinz for a summary of NFR definitions [Gli07]. For the sake of simplicity, if functional requirements define a system's behavior, essentially what a system does; then NFRs define how well that system does what it does. Some NFRs describe a system's constraints and desired qualities. This research is concerned with the following system qualities: usability, performance, simplicity, integratibility, security, robustness and security.

3.1.2.1 Performance

The performance of an interactive system, or measure of that system's ability to support various operations with respect to its utilization of time, computational and network resources, factors very heavily into the resulting user experience. If the functional requirements of a tangible system architecture can be general-

ized to facilitating en-action upon physical embodiments of computational objects and processes and vice versa, then several performance properties (e.g. responsiveness, throughput, bandwidth, latency, etc.) factor into whether the system can achieve these processes within desirable tolerances. For illustrative purposes consider the Tangible Query Interface (TQI)[UIJ03]. Each turn of a parameter wheel results in the generation and processing of a dynamic database query and ultimately a visualization of the query results. 1 TQI, from a high-level view, involves several parallel processes in which multiple parameter wheels can be manipulated, database queries are formulated and processed, and query results are visualized and displayed onscreen. 2 To provide real-time feeling responsiveness, TQI's system architecture must be capable of executing this pipeline from input acquisition to visual responsive in a few fractions of a second (about 100ms). 3 4 The ability of a tangibles interactive systems to execute/support these processes affects overall system performance and user experience. Typical metric considered for system performance are as follows:

- response time
- throughput
- latency

Latency Latency is the measure of time from the moment a payload is sent over a network until the moment in time it is received by its recipient. Latency is a relevant metric for evaluating the performance of a network-based interaction. For instance, network-bound interactions can be no faster than the latencies of network transmission over network links that connect system nodes. So Internet-based applications are constrained by typical latencies on the order of hundreds of milliseconds. When combined with temporal constraints of human interaction (e.g. 100ms for threshold perceived interactivity or 1-2s for threshold human perception of causality), we see the limits of the wide-area network as infrastructure for certain types of classes of interactivity absent strategies such as latency-hiding.

Throughput Throughput is a measure of rate of operations that can be performed or supported over time. The envisioned tangibles-based applications will likely involve processes, which themselves are composed of many simpler processes, processes including the discovery of tangibles, the resolution of the digital content or functionality that associated with a tangible, and the data-flow from user input to software action. A tangible system architecture can be evaluated for various use-cases by examining effect of certain architectural designs on the throughput for various operations.

Integratibility In much of the tangible and embedded interaction-related research, new software has been the emphasis for integration of tangible interaction capabilities. While this approach is beneficial and perhaps necessary for the sake of getting research and development done, some thought on the implications and challenges of integrating tangibles into legacy computer systems. One design challenge of distributed tangible interaction systems is that of building applications and environments that have more pluralistic models of the human-computer interaction than state of the art interactive computer systems typically on top of resources that managed by administrative policies that assume singular human-computer relationships. For instance imagine applications of tangible interaction in which a functionality-rich software suite in engaged not only via mouse, keyboard and screen but also through several physical embodiments of information and capability embedded within that application. Integratability is also an important consideration in the context of preexisting reality-based interaction technologies. Many of these (meta)systems are designed and built with a particular set of requirements and have strengths where other tools do not. To the extent that experiences can be built with different RBI systems that inter-operate across space and time, new tangibles can be designed, built and evaluated more easily.

3.1.2.2 Configurability

A value of tangible interaction techniques is the desire to imbue digital systems with the qualities of physical interaction. One such quality is the flexibility with which one can rearrange and reconfigure one's physical environment, gravity notwithstanding. What is means with respect to tangible interaction is the ease at which one can re-map the bindings between a physical element and any associated digital objects. Reconfigurability is also concerned with how robust a tangible system is to changes such as the sudden absence or introduction of elements. Reconfigurability also plays into how well a system can adapt to various contexts.

3.2 Architectural Views

Kruchten proposes a multiple view model of architecture, to address the needs of multiple stakeholders who may differ in expertise, investment, and interests [Kru95]. Here I use this model to provide a broader view of distributed heterogeneous tangible system architectures. Initially the views presented are intended to be generalizable to a wide variety of tangible applications. Later in this document, I will more narrowly define the architectural features for the domain of distributed visualization tangibles.

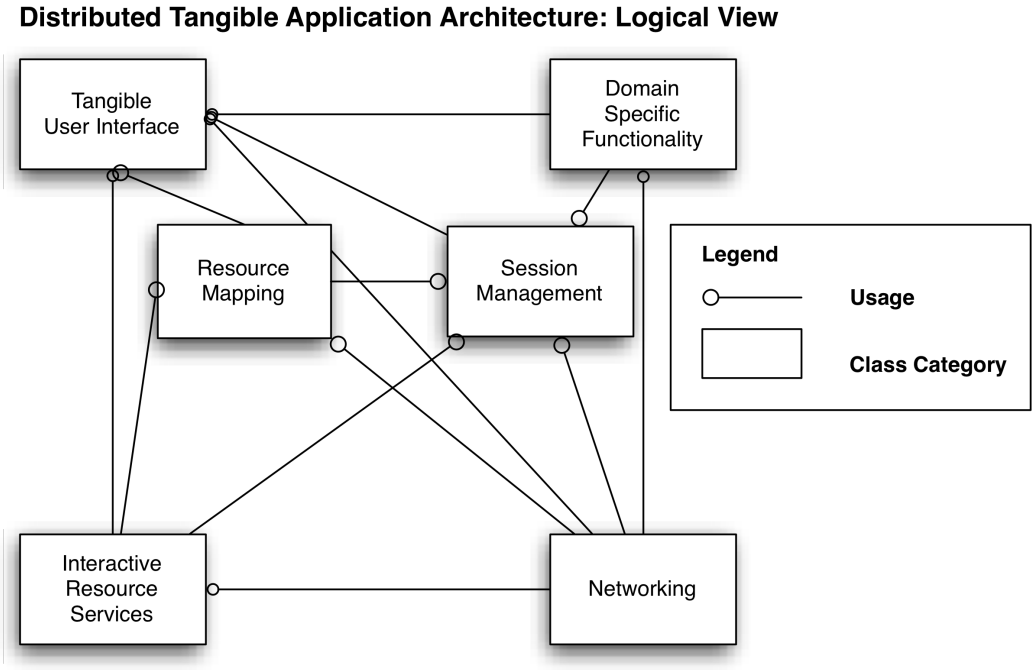

FIGURE 3.1: Tangible System Logical Architecture: The logical architecture defines the structures which support the functional requirements of a system. The logical architecture is organized into classes which provide functionality and services for the domain of tangible interface development and deployment.

3.2.1 Logical View of a Tangible System Architecture

Below are key logical components of a distributed tangible system architecture as illustrated by Figure 3.1

Tangible User Interface These classes define controllers and interfaces to tangible representation within a tangible interface.

Graphical User Interface These classes define controllers and view functional for elements of graphical interface elements.

Domain Specific Functionality These classes provide interfaces to a target application's domain model

Networking Components of this classes provide function for communication and coordination for interactions between entities over network connections.

Interactive Resource Information Services As resources are added to a system the amount of distributed state increases, which may lead to increasing complexity in discovering and utilizing those resources. Interactive Resources Information Services provide mechanisms and interfaces to aggregate resources that may be employed within a distributed tangible system.

Resource Mapping Components of this class provide functionality for binding the state of physical representations to their digital proxies within a tangible system.

Session Management The components provide functionality for managing concerns of user context including which tangible and digital assets are currently accessible by a user. Such functionality becomes necessary when one considers the possibility of many resources being shared across overlapping collections of users and work-flows.

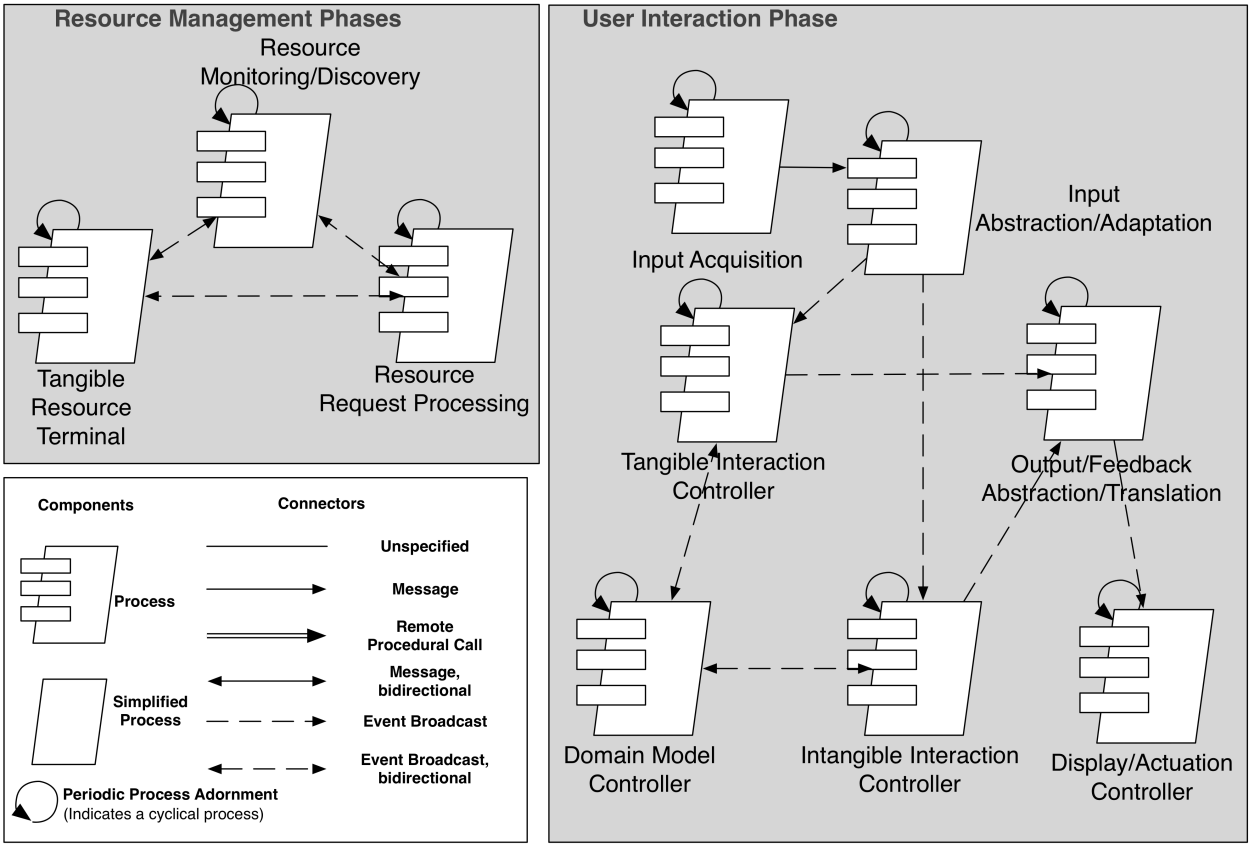

FIGURE 3.2: Tangible System Architecture: Process View Here we see two process clusters, one for the configuration and management of the resources witin a tangibles-based system and the other to represent the process utilizing interactive system resources within a running application.

3.2.2 Process View: Tangible System Architecture

Figure 3.2 illustrates the process view of a tangible interface system architecture. At this time, the two major processes considered for tangibles-based systems centering around the configuration of tangible systems and operation of tangibles-based system. The configuration process consists of interactions between processes for (1) tangible resource discovery and monitoring, (2) resource request processing and (3) a process for viewing and acting upon aggregate tangible resource information. These processes are often not very well

exposed in many local, centralized interactive systems, but when separated into their own modules it opens up opportunities for increasing visibility and simplicity of a potentially distributed system.

The other processes relate to the typical operation of many interactive systems, tangibles included. Theses processes can be further classified into a pair of data flows, one triggered by changes in the physical world in the form of user input or environmental input and other triggered by changes in computational state that result either in some intangible display or actuation of a physical system.

3.2.3 Development View

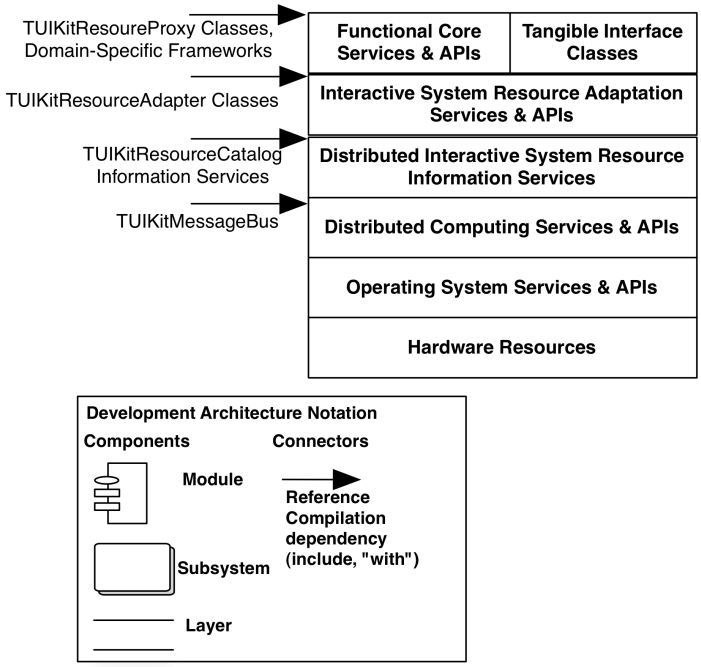

FIGURE 3.3: Tangible System Development Architecture: Illustrated above is a module layout view of a development architecture for tangible interaction with distributed, heterogeneous resources. Each layer provides a narrow, focused API to be used by the layers above it.

This architectural view specifies how software modules are organized within a software environment [Kru95]. Layers are a typical unit of organization within the development architecture. In Fig 3.3, a tangible interaction system architecture is illustrated using a module layout diagram with six layers. Each layer provides a narrow, well-defined interface to the layer above it. The layers that define modules for *Distributed Computing Services and APIs* and *Interactive Resource Adaptation* have been the primary focus of this research as they are needed to provide basic support for tangibles-based applications that employ distributed, heterogeneous resources. By inserting the *Distribute Interactive Resource Information Services* layer, the

resulting architecture can provide services and capabilities to help manage the complexity of configuring and deploying distributed tangibles-based systems of significant scale. An influence for the development architecture presented here is Ganek's models the evolution toward autonomic or self adapting system architectures. While autonomic tangibles-based systems are likely far-off goal, such systems will need to be highly (re)configurable, which is an architectural property of interests for this research [GC03].

3.2.4 Physical View

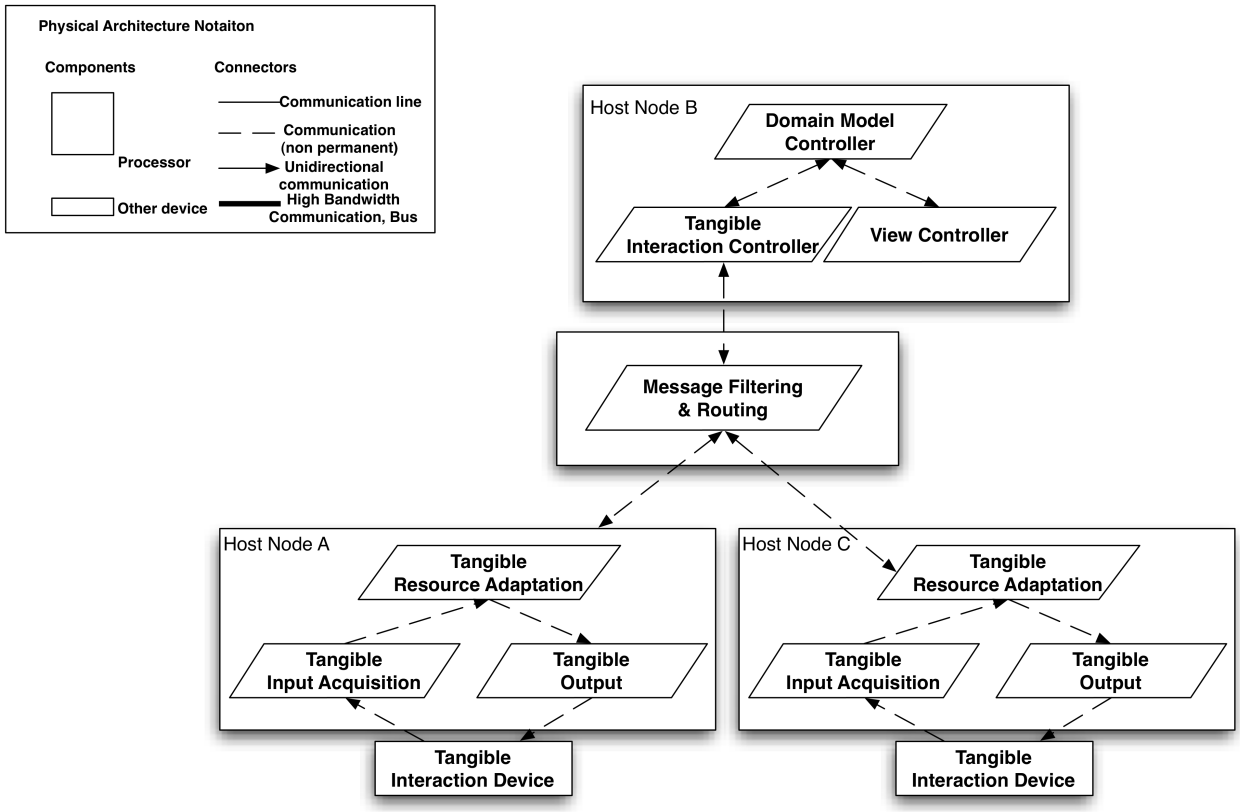

FIGURE 3.4: Tangible System Physical Architecture: An illustration of a partion of tangibles-based system modules across nodes. Within the architecture presented here, the connection between the *tangible interaction controller* process and the *tangible interaction resource adapter* process is bi-directional flow of event and request messages and may be reified as a local or remote connection.

In a view of the physical architecture, we can see properties pertaining to the non-functional system requirements (e.g. availability, reliability, performance, etc) [Kru95]. In this view we can see a partition of processes within an architecture across nodes within a network. In Fig 3.4, we see an illustration of the processes for interactive resource adaptation and integration are partition across two nodes. Fig 3.5

illustrates an example with several nodes and shows how messages are routed and filtered by an additional connector component. These nodes communicate via message broadcasts.

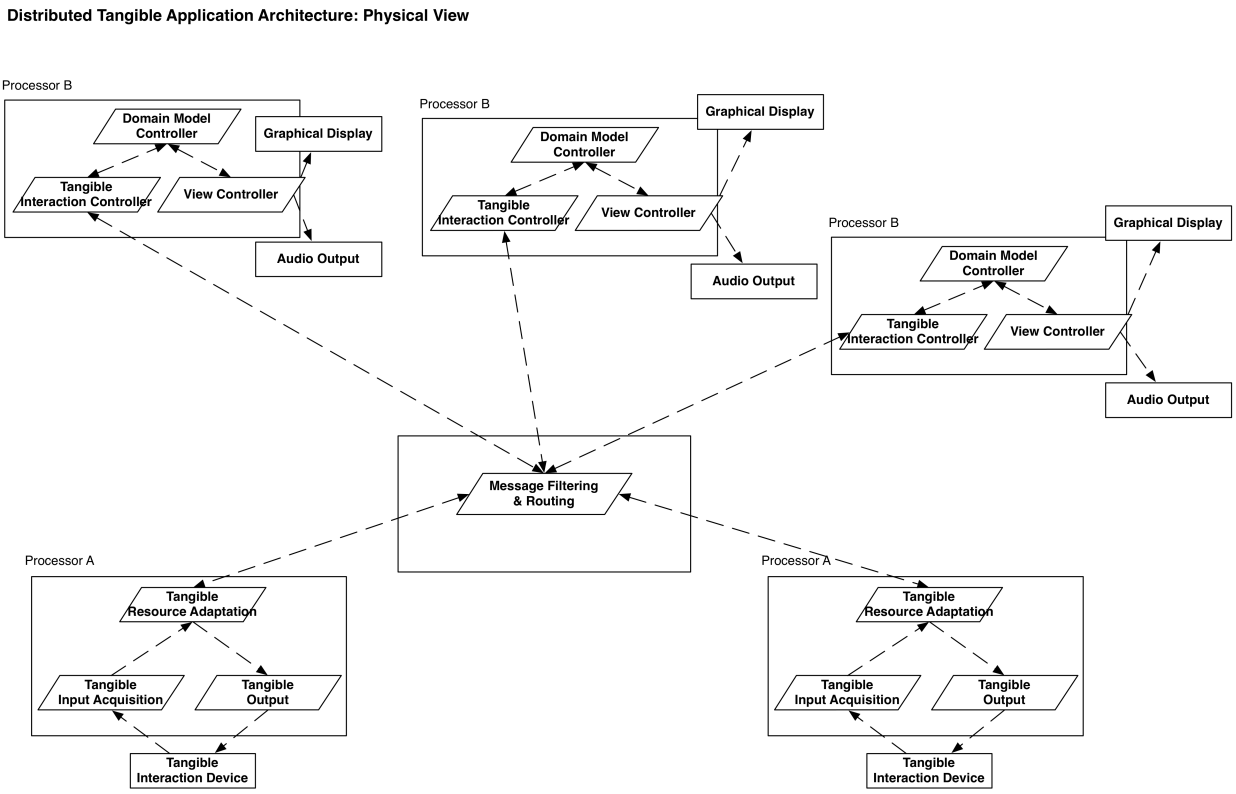

FIGURE 3.5: Tangible System Physical Architecture: Above is an illustration of a physical architecture for a system for tangible interaction in which we have multiple nodes for resource adaption and proxying.

3.3 Architectural Elements

In this section I describe core elements of the proxied tangible interactor architectural style.

3.3.1 Data Elements

The main classes of data elements in the proxied tangible interactor architectural style are resources, events, requests/commands and parameter settings. PTI architectural components communicate by exchanging messages that are encapsulated within a number of number of representations. PTIA resource identifiers combine traditional web architecture resource identifiers with identifiers from the tangible computing domain. See Table 3.1 for a summary of the PTI data elements.

TABLE 3.1: PTI Architecture Data Elements

Elements	Discipline	Example
resource	HCI	RFID reader, tagged object, data visualization services, interaction devices
event	HCI	TangibleSelectionEvent, ResourceAdded, ResourceUpdated, ResourceRemoved
request	software engineering	GetState(), SetState()
representation	web-based systems	HTML document, images, video, audio, any MIME type, JSON object, TUIO message
resource identifier	web-based systems/HCI	URL, URN, TUIKitTopic/ØMQ topic, RFID Tag, NFC tag, visual marker

3.3.1.1 Resources

The PTI architecture is concerned with resources for facilitating physical interaction with digital content. Many of the resources include physical interaction devices or computationally augmented physical objects and environments. Typical devices include but are not limited to:

- radio frequency identification-based (RFID) devices
- physical controls elements (dials, switches, sliders)
- sensors
- actuators
- interaction devices (e.g. pointing devices, special controllers, composite human interface devices)

PTI architecture resources also include objects for the mediation of intangible representations, which include the visual and audio system stacks. These resources usually exist as a multi-granular composition. For instance, in many research systems the physical interaction elements might be exposed at the abstraction level of sensors and actuators or interactors; whereas the graphical elements of such a system might be exposed at the level of widgets or large-grained UI composites. The design challenge of providing powerful, yet high-level abstractions for referencing and utilizing resources lie in striking the right balance between information hiding and specificity.

3.3.1.2 Events

Interaction between PTI architectural components consists primarily of message passing of event representations. The most generic types of events notify that a resources has either been added to or removed from the interactive system context, or its internal state has been updated. Many PTI architecture instances may

define event types for common interactions (e.g. physical manipulation events such as rotation and switch selection).

3.3.1.3 Representations

Unlike the REST architecture style [Fie00], the PTI connector architecture design is stateful. One reason for this that PTI-based interactions are fine-grained and can be high in volume for some applications. Stateful endpoints allow for simpler (e.g. smaller) representations to be exchanged between PTI components. The tradeoff is that the connector architecture is more complex, especially the components in the server-like role. Typical PTI-based interactions involve an application instance wanting to utilize some resource and as such expresses interest in receiving notifications on that resource's state. For some applications, multiple parties may wish to engage a resource both synchronously and asynchronously. Compact representation and communication pattern designs are intended to support interactions between varying numbers and types of components. The form a representation takes in the PTI style often depends on the type of connector facilitating its communication (e.g. web-based connector will use web-based representations, whereas embedded systems-based connectors will use more compact representations).

TABLE 3.2: PTI Architecture Connectors

Connector	Discipline	Example
publisher	networking	paramTray adapter
subscriber	networking	paramTray proxy
forwarder	networking	ØMQ_forwarder
resolver	web-based systems	DNS, key-value-topic resolver

3.3.2 Connectors

Shaw and Clements offer this definition of a connector: A connector is an abstract mechanisms that mediates communication, coordination, or cooperation among components [SC97]. Within the PTI architectural style the primary types of connectors are publisher and subscriber. This is because many of the most common interactions between PTI components broadcast and receipt of notifications, typically events. The publish-subscribe communication model provides a good starting point for implementing a range of communication

patterns (e.g. one-to-one, one-to-many, many-to-many and one-to-any) [JF02]. Additionally, the concepts of adaptation, proxying/delegation and aggregation [1] strongly motivate this design choice.

Because primary PTI component interactions are based on the publish-subscribe pattern, components can be decoupled referentially, semantically, and if you have persistence of data, temporally. This decoupling can be exploited to realize systems that are easily reconfigurable. Distributed tangible applications can be (re)configured during runtime as opposed to be defined a priori at code time. This is integral to the design goal of decoupling the high-level behavior and composition of a tangible from its final reification as a collections of components interacting across multiple digital and physical communication channels.

In addition to publisher and subscriber connectors, the PTI style also includes the forwarder and resolver connector types. A forwarder facilitates communication between publishers and subscribers. For instance, the Zero Message Queue communication library provides a connection-based implementation of publisher and subscriber endpoints. Using only these connectors, components are coupled referentially although it still possible to attain semantic and temporal decoupling (respectively through topic-based publish-subscribe and message queues). A forwarder provide persistent endpoints for subscription and publication of data thus enabling referential/locational decoupling of components. A forwarder can also act as a gateway for communication across multiple domains [2].

The resolver connector, inspired by the REST architectural style, resolves resource identifier into network addresses and topics for inter-component connection. A preexisting and common example of a resolver used in PTI architectures is DNS, as many resources utilize the legacy Internet and Web networking stacks. A new type of resolver introduced within the PTI style is one that resolves a more human-readable topic into the a topic of filter for a particular implementation of the publish-subscribe communication pattern [3]. A PTI architecture might also incorporate client-server connector, particular RESTful or Web-based connectors.

A publisher connector sends notifications to one or more subscribers that have subscribed to that publisher's topic [4] . A publisher connector can also be used to make requests on resources; but because publish-

[1] [*Adaptation* is concerned with unifying interfaces of diverse resources with similar behavior or capability. *Proxying and delegation* address how local computation is bound to some state of remote resources. And through *aggregation*, users are able to form logical compositions of tangible computing application from disparate (physically and technologically) resources.]

[2] [When combined with channel agnostic connectors and local and remote communication can be bridged. Also the forwarder can be federated to realize scalable, decentralized interaction between PTI components.]

[3] [For instance 0MQ's pub-sub uses a simple string for message topics. This could more easily be used for defining hierarchies or one dimensional namespaces. But for n-dimensional namespaces, a resolver is needed to flatten a multidimensional topic identifier and perhaps save the users from having to know a strict order of identifier parameters.]

[4] Current implementations of PTI style used a connection-based, topic-based style of publish-subscribe. However, it reasonable for PTI architectures to be implemented using other approaches to publish-subscribe.

ers and subscribers are decoupled, additional components and logic are needed to facilitate the request-reply interaction [5] . This publish-subscribe approach to request-reply is more complex than a client-server-based approach for one-to-one communications, but may lead to simpler designs for request-reply semantics involving one-to-any and one-to-many interactions.

The subscriber connector receives notifications and invokes methods on the proxy components that are mapped to a particular resource. Analogous to the use of publisher connectors for request-reply-based interactions, subscribers can be used correspondingly. The most basic programming task for PTI-based applications is the definition and attachment of notification handlers via a resource proxy interface. The interface for the resource proxy component, which will be described in more detailed in the next section, consists of notification handlers for events, requests, and replies.

3.3.3 Components

The basic premise of the PTI Architecture style is that applications are collections of abstract resources proxies. A proxy may be bound to one or more concrete resources. This binding of a proxy to its concrete resource is achieved through the connection of a proxy to an adaptor. Depending upon the publish-subscribe connector implementation, a resource proxy and adaptor can interact over both local and physical channels, meaning that a aggregate of PTI-based application could execute over local or distributed resources. The uniformity of the resource proxy interface combined with its flexibility in the number, locality and class of concrete resource adaptors to which it can be bound, means that a user can define a PTI application kernel, consisting of one more resource proxies can be executing across a variable collection of concrete resources that can vary in the number, locality and technical underpinnings.

The primary components within the PTI architectural style are resource adapters, resource proxies, and resource catalogs.

3.3.3.1 Resource Adaptors

Resource adaptors intermediate for PTI resources. This components get its name from its role in adapting an implementation specific interface for a particular resource into an abstract interface for resources with similar structure and behavior. Adaptors are responsible for collecting data from or about a resource and

[5] A topic-based pubsub system can support request-reply interactions by using topics as identifiers. For instance, a component can use its publisher endpoint to make a request, supplying a topic to which is subscribed to receive the corresponding reply.

TABLE 3.3: PTI Architecture Components

Component	Discipline	Example
resource adapter	design patterns	paramTrayAdapter, vizAdapter
resource proxy	design patterns/HCI	paramTrayProxy, vizServiceProxy
resource information provider	web/grid services	any resource proxy or adapter
resource information services	web/grid services	TUIKit catalog

notifying interested parties of that resource's state. Adaptors also receive requests on a resource's behalf and invoke the appropriate implementation-specific command or precedure.

3.3.3.2 Resource proxies

Resource proxies are responsible for receiving notifications on the resources to which it is mapped and invoking the appropriate application-specific procedure to handle the data encapsulated within that notification. Resource proxies also provide an uniform interface to one or more concrete resources, thus enabling a user to utlize heterogeneous resources through a single interface.

3.3.3.3 Resource Catalogs

Resource catalogs provide an interface to access the status of a collection of PTI resources. As mentioned earlier, many PTI interactions are event-based. While this makes it easy to configure a system with potentially many moving parts, it becomes difficult for users/developers to have a broader picture of the entire system. Some external component is needed to monitor the a running EBI-based system. A resource catalog serves that role within the PTI architecture style.

3.4 Summary

In this chapter discussion has been given to software architectures that support TUI-based applications in distributed, heterogeneous environments. It began by reexamining questions central to this dissertation in terms of distributed, heterogeneous tangible systems approaches found in the tangible, embedded and embodied interaction (TEI) literature as well as concepts of software engineering.

This chapter went on to explore these questions by first examining the functional and non-functional requirements of systems that support distributed, heterogeneous tangibles. Some functional requirements for such systems are as follows:

- to support the design, development, deployment and refinement of tangible interface elements;

- to support tangible interaction hardware access;
- to support tangible input acquisition;
- to support tangible output dispatch;
- and to support association of digital state with physical state.

If functional requirements define the behavior of a system in terms of the what it does or the services it provides, then non-functional requirements specify how a system must do it. Distributed, heterogeneous tangible systems requirements are as follows:

Performance For fine-grained interaction, the ability to support event-based operations at a rate greater than 10 operations per second.

Scalability For fine-grained interaction, the ability to support 10 operations/second as the number of system nodes increases.

Extensibility To support the ability to integrate novel resources (e.g. interaction devices, application domain-specific components and services, etc.) and resource classes.

Configurability To support the ability map and remap the bindings between abstract specifications of a tangibles-based system and the runtime resources that are used to execute that specification.

To address the several stakeholder classes of a TUI-based system lifecycle, this chapters provides specifications of the PTI architecture that are based on the 4+1 View Model of Architecture. This chapter then goes on to describe the PTI architecture in terms of the its basic elements which include data elements, components and connectors. The primary interaction within the PTI architecture are between adapter components which broadcast events on resources to a number proxies over publish-subscribe channels. The publish-subscribe communication model decouples (spatially and synchronously) resources within a TUI from the application level-components that may utilize those resources. The semantics of the adapter-proxy communication is defined by generic APIs act as wrappers for number of diverse implementations that exhibit similar behavior or functionality. This architectural design is informed by the disciplines of TUI-based systems and frameworks, software engineering, and distributed and web-based computing.

Chapter 4

TUIKit: A Software Development Toolkit for Tangible Interaction with Distributed, Heterogeneous Resources

In the previous chapter the PTI architectural style is introduced. TUIKit, as described in this chapter, implements a subset of the architectural features of the PTI style. TUIKit was designed to support the development of tangible interaction-based applications which may be composed of distributed, heterogeneous resources by enabling, its end-users, tangible interface developers, to abstract details such as the number, location and implementation type for resources that may be used within a tangible interaction-based system. In the remainder of this chapter discussion is given on the architecture and implementation of TUIKit.

4.1 The TUIKit architecture

Motivated by experiences in building tangibles to support visualization as well as supporting novice interface developers in prototyping tangible interfaces, we began work on what would become TUIKit. A motivating question was, ”How could one code or specify an object’s logic within an interactive system despite many different implementations?” Would be possible for an object to advertise its behavior as an interface and be pluggable within a ecosystem of other tangible interaction system resources? And what about the pragmatics of communication between the various components of which an tangible interaction-based system might comprise. These questions from a systems perspective, related to the and heterogeneity and distribution across data networks of resources within an tangibles-based system. The design intent of TUIKit is to provide abstractions and tools to enable its users to exploit heterogeneity and distribution to realize scalable tangibles-based applications.

The software architecture of TUIKit, as it exists now, is message-based. The fundamental assumption of applications built with TUIKit is that they are loosely coupled logically distributed by default, even if all software components in a TUIKit-based application are deployed locally. This is made possible through the use of a messaging application programming interface (API) and protocol that abstracts the physical channels over which messages are transported. Furthermore this abstractions allows TUIKit components to be deployed and redeployed across system boundaries with minimal changes to their core logic.

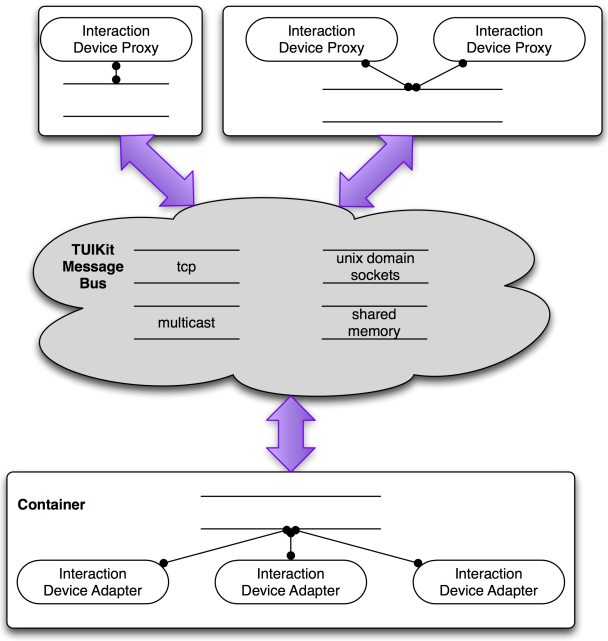

FIGURE 4.1: Module Layout Diagram of the TUIKit Architecture

The TUIKit architecture consists of three layers: a tangible computing resource layer; a tangible computing proxy layer for application-level integration; and a messaging layer to support communication and coordination of components among and between resource and proxy layers. In next three subsections, each layer will be discussed in more detail.

4.1.1 Resource Layer

The primary function of the resource layer manage communication with resources that may be utilized within a tangibles-based application. Within a tangibles-based application, functionality for input acquisition and output display is also provided.

The primary type of component within the resource layer is the resource adapter. As mentioned in §3.3.3.1, resource adaptor intermediate for tangible computing resources within PTI architectures. A resource adaptor monitors the state of a tangible either directly(e.g. it might serve as the hardware driver for an interaction device) or indirectly (i.e. monitor the state of a resources via communication with some intermediary for a resource).

There are primary two types of resource adaptors, input and output adaptors. An input adaptor transforms state changes on an input resource (e.g. sensors, tangible devices, etc.) into messages corresponding to events that constitute behavior for a class of tangibles and broadcast those messages on the Interac-

tion Message Bus, the TUIKit connector. An output adaptor listens for request messages corresponding to methods within the interface for a particular class of tangible computing resources, invokes the appropriate implementation specific method or procedure for that resource, and transforms that request's response into the appropriate format for transmission to the original requester or broadcast to interest parties. Because of the publish-subscribe architecture of the connector interface and the design intent of facilitating system composition without requiring identity, a resource adaptor is not responsible for ensuring that a requestor receives a response to their request. This is will be handled by the connector interface. [1]

4.1.2 Resource Proxy Layer

At this layer within the TUIKit architecture, we find components for application-level integration. To support migratability of resources within a distributed tangibles-based system, the user does not directly access the resource through its interface but rather through a proxy. The use of the resource proxies enable us to hide from the user the location of and specifics about the implementation of that resource provider. Currently the main component types are the ProxyController and InteractorProxy.

Resource proxies are components which act on behalf of concrete resources and are integrated at application code-level[2]. These proxies can be bound (and rebound) to concrete resources at runtime. So for instance, one might want to integrate RFID-based object recognition, but through use of a generic tag reader proxy could integrate computer vision-based object recognition. It is also possible to bind multiple concrete resources to the same proxy, which could be exploited for basic support of multimodal interaction.

4.1.3 Messaging Layer

The TUIKit Data Bus provides API, services and protocols for communication between TUIKit components. As mentioned earlier, TUIKit is primarily a message-based architecture[3]. We currently use the Zero Message Queue (0MQ) message library[4], which provides socket-like interfaces for communication over multiple physical channels (e.g. TCP connections and multicast for network-based communication, filesystem for inter-process communication and shared memory for intra-process communication). More specifi-

[1] The simplest way to implement request-reply interactions with publish-subscribe-based connectors, would be include a requestor field in the request message, likely a requestor-specific topic, and have the adaptor simply publish, a response on the topic specified by the requestor.

[2] TUKit applications are currently built as applications built with the TUIKit class library, but in the future TUIKit applications could be realized through composition of services and use alternative development and runtime environments such as visual dataflow workbenches.

[3] In the future, we would also like to support other communication mechanisms such as streams.

[4] http://zeromq.org

cally TUIKit components communicate over several publish-subscribe channels, thus enabling the physical topology of a system to be defined at runtime.

TUIKit provides a message interface for the definition, encoding and decoding of interaction events and resource request messages that exchanged are between TUIKit resource adapters and proxies. Currently messages are encoded in the javascript object notation (JSON) format, which was chosen for its cross language support and human readability as well for the possible of integration within web-based environments. In the future we plan to add support for encoding/decoding TUIKit messages as TUIO[5] messages. For users of the application-level library, this detail is hidden.

4.2 Implementation

TUIKit consists of device drivers and plug-ins, a device manager, an interaction message bus and class library with bindings for python, Java and C#. TUIKit currently supports several mechanical-electronic (mechatronic) interaction devices including dials basedupon several types of sensors and an RFID reader, which were all developed using the Blades and Tiles modular interaction hardware toolkit [SUR+09]. This toolkit also supports commercially available physical interaction devices including the Griffin Powermate media control knob, and RFID modules by ID Innovations and Olimex. For the Surface Oil Spill application, we wrote an adapter to transform OSC messages generated by the touchOSC iphone/iPad application. We are currently preparing TUIKit for release as an open source library.

The code snippet in Fig 4.1 shows how one might integrate a rotary input to control the zoom camera function within a visualization application. Early versions of the TUIKit class library were used in two semesters of an introductory interface design and technology course. This library was used for integrating physical controls and tangibles with existing graphical user interfaces, and so there was a desire to integrate into programming environments familiar to the students. At the time of this writing, TUIKit has been used to integrate tangible input into applications built with Java SWT, Java Swing, Processing/Java, OpenGL/C++, and C# /Windows Presentation Frameworks (WPF).

TUIKit builds upon the innovations of many prior and parallel efforts for building applications that employ post-WIMP interaction techniques. Ultimately, we wish to realize system architectures capable of scaling in the number of users, devices and locales. In the future we will perform performance tests to evaluate

[5] http://tuio.org

```
class vizController:
    #Zoom camera when dial is turned
    def zoomOnDialEvent(self, event):
        delta = event.value - self.last_val
        self.zoom(delta)
def main():
    ixerCtrl = interactorController()
    dial1 = ixerCtrl.getInteractor('rotary1')
    vizCtrl = vizController()
    dial1.addListener(TUIKit.ROTATEDIAL,
        vizCtrl.zoomOnDialEvent)
    ixerCtrl.add(dial1)
    ixerCntrlr.start()
```

Listing 4.1: Code Snippet Illustrative of How to Integrate Physical Dials to Control Visualization Software

how scalable TUIKit is. Our goal is to achieve responsiveness of at least 10 events per second per device (100ms) and low jitter for interaction over high latency network connections. We think this is possible because the interaction message bus is based on communication infrastructure designed to handle 1000s of message per second [zer10]. The challenge will be providing developers with abstractions that allow them to more easily design and manage such large systems.

4.2.1 A Note on Communication Mechanisms for Distributed Interactive Systems

In choosing communication infrastructure on which base our interactive systems, we wanted to strike the right balance between high performance and ease in constructing and managing a running system. We have come to evaluate communication protocols and infrastructure in terms of their performance, support for high-level communication patterns/models, pragmatics of deployment and integration.

The BSD Unix sockets API provides the most widely available, basic functionality for network programming. Because it is a relatively low-level interface, you can achieve high performance communication but at a high cost in complexity. Alternately, even reasonably simple networked applications can have poor performance with poor socket-based programming.

At the other end of the spectrum we have higher-level communication mechanisms such as shared data spaces-based architectures (e.g. EventHeap and EQUIP2). Johanson and Fox enumerate the various communication patterns needed to build ubiquitous computing systems (i.e. one-to-one, one-to-many, many-to-many and one-to-any). They go on to compare various types of communication and coordination infrastructure technologies. These are client-server, tuple spaces, remote procedure call, message oriented middleware and publish-subscribe. They state that tuple spaces support more of these communication models by default

TABLE 4.1: Comparison of Communication Infrastructure

technology	type	external dependencies	1-1	1-many	1-any	many-many	usability	broad support	performance
sockets	client-server	-	✓	✓	-	-	+	++++	+++
XMLRPC	client-server	+	✓	-	-	-	++	++++	-
eventheap	tuplespace	++	✓	✓	✓	✓	++	++	++ [a]
ZeroMQ	publish-subscribe	+	✓	✓	-	✓	+++	+++	+++
ICE	RPC	++	✓	✓	-	-	++	+++	++
TUIO	client-server	+	✓	✓	-	-	+++	+++	+++
SharedPhidgets	spaces-based	++	✓	✓	-	✓	++++	+	++

[a] high performance for few nodes on a local network

and so chose it as the basis of EventHeap, the infrastructure underlying the Interactive Room Operating System (iROS) and later iStuff. We decided not to adopt the EventHeap technology, but very much look to the ideas inherent in its design.

Another dimension of the communication infrastructure is concerned with the semantics of data exchange between system components. With tuplespace-based infrastructure the meaning of messages are defined by their type. With RPC systems, communication is defined in terms of method calls and returns values, which are ultimately defined in terms of the client language type system. This type of infrastructure maps well to operations involving resource requests. Other transport protocols define no such semantics thus leaving the freedom to define those semantics. OSC is a presentation layer protocol on top of which several protocols have been defined for various types of spatial tangible interfaces (e.g. TUIO and LusidOSC). In terms of number of implementations for various interaction devices, languages and software platforms, TUIO is perhaps one of the most widely adopted. For this reason we seriously considered adopting it for the initial versions of our toolkit; but because most OSC implementations and thus TUIO are tied to client-server communication-based protocols, we decided to delay adoption of TUIO until its semantics could be implemented on top of transport protocols that support higher-level communication patterns such as publish-subscribe or multicast.

The pragmatics of integrating communication infrastructure with interactive software components also factored into selection of protocols and infrastructure. Using operating system-native sockets API would incur the least overhead in terms of external dependencies because it is supported natively by most modern operating systems and programming languages. On the other hand, implementations would likely be very complex for systems with large or varying numbers of components. Sophisticated distributed application development frameworks such as ZeroC ICE and many message-oriented middleware incur dependencies on large code libraries and/or infrastructure such as message brokers. Additionally, these frameworks can also lessen flexibility in the structure of software that can be built with them. [6]

Another pragmatic concern is the coupling between system components. With point-to-point communication mechanisms, a component has to refer to every endpoint to which it intends to communicate. This makes implementing one-to-many communication patterns difficult. This tight coupling also intro-

[6]For instance, the python implementation of ZeroC requires ICE infrastructure must be initialized in the main function of your program. Also the method to call listen for remote method invocations is a blocking call, which interferes with the events loops of most interaction application frameworks.

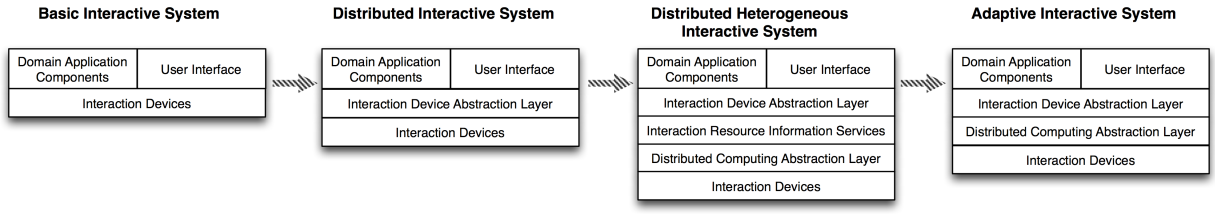

FIGURE 4.2: Evolution toward Adaptive Systems for Tangible Interaction with Distributed, Heterogeneous Reources

duces strict ordering in component life cycles and possible system-wide fail states when single components go offline. Mechanisms that provide publish-subscribe communication patterns decoupling senders from receivers. And so depending on implementation, the topology of a distributed interactive system can be changed by vary how modules publish and subscribe data. Systems like iStuff's Patch Panel further decouple interactive system components by enabling their event driven behavior to be defined in terms of behavior belonging only to that component [BSF04]. This enables spontaneous integration [KF02].

4.3 Related Work

This work builds upon the efforts of several in the tangible interface toolkits and frameworks community. In this section we will overview this work using the framework that is based on an evolution of interactive systems from simple, homogeneous types to those that can adaptively incorporate distributed, heterogeneous resources.

We look to IBMs autonomic deployment model, which describes the incremental addition of architectural layers and features for the evolution toward self-managing systems [GC03]. In our model the first stage represents basic interactive systems. In Figure 4.2 we see that major class of components in such systems are interaction devices, functional core or objects that provide domain-specific functionality, and user interface components. With respect to tangible user interfaces, there were several frameworks which provided abstractions and protocols that make it easier to build systems of this type [GF01, VG07, HYA+08, SUR+09]. These toolkits often focus on providing interfaces to post-WIMP interaction devices to support physical user interface prototyping and implementation.

By adding abstractions that support access to interaction devices in implementation agnostic manner, applications can more readily integrate diverse post-WIMP interactive resources. These abstractions and protocols enable interface developers to realize an interface with multiple interface component technologies

which are related by key characteristic (e.g. both computer vision-based and radio frequency identification (RFID)-based sensors detect the presence of an object within a tangible interface). Several physical interaction frameworks provide such abstractions [KL09, BRSB03, KBBC05, KTK+05].

In the next phase, we see the addition of abstractions and protocols for communication between and coordination of components that comprise a tangible interface. Key features/aspects of this abstraction layer include transport protocols, data interchange formats/protocols, information services and application programming interfaces. Interactive system architectures which provide abstractions of this type make it possible to realize interfaces that are composed of components across multiple address spaces. Such systems support contexts such as teleoperation and remote distributed collaboration. In the research areas of tangible interaction and related ubiquitous computing we find several frameworks that provide such abstractions and are generally targeted at allowing interface developers to focus less on network programming and more realizing their interfaces [BRSB03, Hol09, JF04, KBBC05, MG07].

As tangible interaction systems grow in complexity (in terms of the number, type and locality of interface components), an information abstraction layer provides interfaces for accessing the aggregate state of resources that may be comprise a distributed, heterogeneous interface. The information collated at this layer can also supports the automated configuration of interactive systems. With some-event driven tangible application frameworks interaction device resources are exposed through the events these resources either trigger or react to [BRSB03, KBBC05, KTK+05]. With this approach, UI composition and coordination is defined declaratively through the definition of interests in certain event types. Other frameworks do provide abstractions for explicitly querying and allocating distributed interface component [DLG+08, KTK+05, MG07]

4.4 Strengths and Weaknesses

The proxy tangible interactor (PTI) architecture was devised to allow more comprehensive examination of distributed, heterogeneous systems for tangible interaction that may not exist in full for years to come. It was also derived as a way to guide the development and evaluation of TUIKit or subsequent implementations of an PTI architecture. In light of this, here I use the description of PTI-based architectures to discuss the strengths and limitations of TUIKit.

In Table 4.2, we see what elements of the PTI architecture are supported by current TUIKit implementations. From these unsupported elements, we can see what interactions are supported by applications built

TABLE 4.2: PTI Architectural Features Implemented by TUIKit

Element	PTI	TUIKit
adapter	✓	✓
proxy	✓	✓
publisher	✓	✓
subscriber	✓	✓
message	✓	-
event	✓	✓
request	✓	x
reply	✓	x
information provider	✓	x
resource catalog	✓	x
configuration interface	x	x

with TUIKit. For instance, the TUIKit-based systems do support request-reply semantics, which can be seen in the lack of an implementation of message types for requests and replies. Another limitation is inherent in the decoupled nature of TUIKit component interaction, which is influenced by the use of a publish-subscribe-based connector and event-based invocation (EBI) style for peer-to-peer communication amongst TUIKit-based components. Both these styles result is systems that are highly composable and evolvable, yet such such have low visibility and are difficult to debug as a non-running system [Fie05]. In consideration of these limitations, in the PTI architecture higher-level components and services are specified for managing the entire system. At this level, it becomes possible to examine the effects of a local change upon a larger (sub)system.

TUIKit-based software has a high degree of evolvability. Ciraci and van den Broek define evolution from a systems architecture perspective as “changes in a system’s environment(domain), requirements(experience) and implementation technologies(process)” Accordingly, evolvability is the degree to which a system can survive changes to its environment, requirements and implementation technologies [CvdB06]. At the time of writing this document, TUIKit has only limited external use, and so evolvability in terms of changes to evironment and requirements will have to explored later; however, it is possible to consider how evolvable TUIKit-based systems are with respect to implementation technologies. An original design intent of TUIKit was to support the development of tangible user interfaces through the use of the multiple implementation technologies. To the extent a resource for tangible interaction exhibits behavior similar to the interface of an existing adapter, an existing TUIKit application can be mapped onto that resource. And because TUIKit

is a message-based, data centric architecture, it is straightforward to implement TUIKit in for programming languages, operating systems and application frameworks.

TUIKit is designed to be extensible. For instance, new classes of resources can be supported by TUIKit-based applications by implementing an abstract adapter and proxy for that resource class. Support for integration of specific instances of a resource (e.g. a tangible input device, visualization software service, etc.) can be added to implementing an adapter instance for that resource.

4.5 Summary

TUIKit supports development of applications based on the PTI architecture and consists of a resource layer and resource proxy layer which are connected via a lightweight messaging layer. TUIKit has implementations in several client languages and was used to support project development within an introductory interface design and development course. And while TUIKit is inspired and influenced by prior and parallel research efforts towards toolkits and frameworks for tangible interaction-based systems, it is distinguished from them through its architecture which supports interactive systems for fine-grained interaction using a range of resources.

The core design idea behind TUIKit is to facilitate flexible communication between the software that defines the structure and behavior of a tangibles-based application and the resources that support that application. This is achieved by using a communication layer that abstracts physical communication channels from the logic of the communication components, as well as providing high-level communication patterns such as publish-subscribe, which makes the topology of the distributed tangibles-based system configurable. In order words, TUIKit application components can employ remote resources just as easily as local resources. Through use of resource adapters and abstract interfaces for resource classifications with similar behavior or functionality, TUIKit-based applications can integrate diverse resources. And through flexible binding of application-level components (proxies) to resources via one or more adapters, a TUIKit-based application can be configured to vary in the number, type and location of resources within that application.

Chapter 5

Discussion

An architecture to support TUI-base applications in computing environments with distributed, heterogeneous resources was presented in Chapter 3. This architecture was further explored in Chapter 4 through TUIKit, a toolkit for developing TUI-based applications with distributed heterogeneous computing resources that embodies a subset of the design principles defined by the PTI architecture. In this chapter, discussion is given to architectural and technical properties of the research presented in this dissertation. As mentioned earlier, a design goal for this architecture was to provide abstractions that allowed a high-level specification of a TUI-based application to be mapped to configurations that may vary in the number, type and locality of components of which a tangibles-based system is composed.

In the next section (§5.1), PTI is described in terms of the architectural style it comprises. The lens of software architectural styles is used as a basis for the evaluation of PTI. It is difficult to analyze architectures because they are often presented as a whole, which often obscures the design constraints and rationale of a particular architecture. Architecture styles also provide a tool for the objective comparison and evaluation of software architecture [Fie00]. After describing basic architectures relevant to software architectures for distributed, heterogeneous tangible interaction, the derivation of PTI's architectural style is given. We then use a classification of architectural styles to contrast and compare PTI against other architecture for distributed, heterogeneous tangible interaction.

5.1 Architectural Styles for Distributed Interactive Systems

Shaw and Clements define an architectural style as a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with the local or global constraints on the way composition is done [SC97]. The goal of this section is to examine the architectural properties of the PTI architecture at the center of this dissertation's thesis. Architectural styles define a standard language for describing architectural compositions. Styles also enable us to decompose an architecture in substructures and consider the architectural properties, constraints and design rationale for each of those structures. The systematic decomposition of software architectures also supports comparison and contrast-

ing of diverse architectures by providing common ground and objective measures for evaluation of various architectures. For instance, the TUIO/ReacTIVision system incorporates a client-server hierarchical style [KB07a]; given that client-server architectures promote scalability, simplicity, and evolvability [Sin92], it reasonable to posit that TUIO-based systems are scalable, simple and evolvable to the extent that other architectural design elements don't reduce these properties.

In the following subsection, a brief survey of architectural styles relevant to interactive systems with distributed, heterogeneous resources is given. Following this survey, this chapter details the derivation of the PTI architecture in terms of the styles it comprises. Based on classifications of software architectural styles by Shaw and Clements as well as Fielding, the PTI architecture is compared and contrasted with other architectures for interactive systems supported by distributed, heterogeneous resources.

5.1.1 Data-flow Styles

5.1.1.1 Pipe & Filter and Uniform Pipe & Filter

In the pipe and filter(PF) style, components, called filters, operate on input data and produce output data to be passed onto potentially some other component. This data is passed between filters via connectors called pipes. Common applications of the pipe and filter style include signal processing and distributed systems.

The pipe and filter (PF) style has several advantages. One advantage of PF styles is simplicity. In the pipe and filter style, filter components are independent of other filters. This independence of filters promotes extensibility. Also, filters do not know the identity of upstream nor downstream components. The PF style also promotes reuse, as new systems can be composed by combining different types of filters. On the other hand, one disadvantage of PF-based architectures is a tendency toward the batch processing, which is often not a good fit for interactive systems that require incremental updates [GS94].

Many implementations of the TUIO protocol used the pipe and filter style [tui11], in which pattern recognition components, typically regular expression scanners, filter TUIO input streams and invoke message handlers that have been assigned to a particular pattern. Another application of the PF style in distributed, heterogeneous interactive systems is iStuff [BRSB03]. The iStuff system provices a component, called PatchPanel, that enables one to patch together devices by defining filters that transform events from one device into signals on one or more other devices.

5.1.2 Hierarchical

5.1.2.1 Client-Server

In the client-server(CS) architectural style, the components exist within a hierarchy in which client components make requests of server components. This is perhaps the simplest architectural style to implement among network-based RBI systems, although several distributed RBI architectures adopt a shared data architectural style over the client-server architecture style [BRSB03, GIM+04, JF02, KTK+05, MG07]. One explanation for this is that many interactive systems, especially those designed with the intent of mediating interaction among many objects, are event-based and not a high impedance match for the request-reply interactions within of a client-server architecture.

5.1.2.2 Layered System and Layered Client-Server

A layered system is a style in which components are organized into a hierarchy of layers in which components provide services to components in the layer above it and utilize services provided by the layer below it. One of the most widely known layered system architectures is the OSI System Model for Networking Applications [Zim80]. One purpose of this style of is to promote evolvability of the larger system and reusability of the existing code. Evolvability is improved by use of this style because components of each layer are simple because they need only access low-level services via interfaces from the layer immediately below it. This constraint decreases coupling among modules from non-adjacent layers. Reusability is promoted by the layered system architectural style because upper layer modules can take advantage of lower-level capability (especially in the non-adjacent layers) without having to explicitly access those lower-level APIs.

Shared Phidgets and several TUIO-based application frameworks are examples of architectures based on the layered system style [MG07, KBBC05, tui11]. Shared Phidgets employs the layered system style to provide mechanisms that simplify distributed user interface development by progressively hiding details of network programming. Common among several TUIO-based application frameworks is the layering of services, APIs and protocols from a particular application development framework atop components that transforms TUIO-based input into native input formats [tui11].

Although the layered system style is considered to be a “pure style,” as noted by Fielding [Fie00], for network-based applications this style is often paired with the client-server style to create a layered client-server style. Spatial tangible interaction systems that are based on the TUIO protocol have essentially lay-

ered client-server style architectures. Shared phidgets represents an architecture in which layered style is combined not with a client-server hierarchical style but peer-to-peer style as exemplified by the Shared Dictionary subsystem [MG07, Mar08].

5.1.3 Peer-to-Peer Styles

5.1.3.1 Event-based Integration

The event-based integration(EBI) style enables components to interact without requiring identity. This style is also known as the implicit invocation style. Typical interactions involve some components broadcasting events on a shared event bus, while other components express an interest is some events and provide methods or procedures to invoked to handle such events. The design intent behind this style is to reduce coupling. The Model-View-Controller architectural pattern, and thus many subsequent interactive systems with similar structures, incorporate the EBI style.

The EBI style has strengths in 1. support for extensibility through the ease of adding new components; 2. reuse through the tendency toward general event interfaces and integration mechanism; 3. and evolution by enabling components to be replaced without requiring changes to the interfaces of other components. EBI style architectures may have issues with scalability and reliability because of the reliance on a single event bus. These weaknesses can be address through the use of layering and filtering but at the cost of simplicity.

5.1.3.2 C2

The C2 architectural style was purposed for supporting large-grain system reuse and composition in GUI-based software that goes beyond the creation and use of GUI toolkits [TMA+96]. This style is derived from a combination of the EBI and client-server style. The C2 style is characterized by components which reside in layers and are constrained to only see components above it within the system hierarchy. Components interact by asynchronously passing messages of two types: notifications and requests. Notifications travel down within the system, thus enabling, lower-level components to be unaware of the higher-level components; and requests are sent up to higher layers. Taylor et al. provide an example in which a application provides an audio-visual interface for stack manipulation [TMA+96]. In this example architecture for a GUI-based the applications, the bottom layer would consist of interaction devices, while the top layer would contain application code. Furthermore, when a user manipulates an input device and generates a user

event, a message is sent up from the device layer requesting some component from a layer above to invoke some computation. When a user's action results in the modification of an application's model, a notification of that change is sent down to be handled by some lower-level components (e.g. display manager).

The C2 addresses the scalability issues of EBI style architectures by organizing components into layers and the adding message filtering, which also improves evolvability and reusability. Heavyweight connectors that provide monitoring can improve an architecture's visibility and reliability. C2 is intriguing as a style in that it essentially defines a meta-architecture for describing or specifying other architectures. Another property of the C2 style is the separation of the conceptual architecture from the implementation architecture [TMA+96]. For instance, while a constraint of the C2 style is that each component exists within its own thread of control, the final system architecture may require modules to be grouped within the same process for the sake of performance or some other design goal.

5.1.4 Connector Styles

Connectors are architectural elements that support communication and coordination between components within an architecture. Connectors act as mediators between components. In the software architecture styles literature, connectors are not typically treated as individual styles, but the choice of connector architecture can greatly influence the architectural properties of a systems. For instance, constraints imposed by a connector's control structure affect the manner in which a particular connector style can be integrated into an existing architecture. This is especially the case with large scale distributed systems for tangible interaction that will likely incorporate several types of connectors to create higher order mechanisms for communication and coordination of components within a system. For these reasons, this dissertation examines several connector styles that have been utilize by TUI-based systems in the literature.

5.1.4.1 Client-Server

Client-server (CS) connectors facilitate interaction between components is based on request-reply semantics. Through a client port, a component request a service that is provided by some server component. The client-server implementation defines the media and protocol over which request-reply semantics are realized. Client-server connectors typically are implemented via point-to-point connections (e.g. TCP, Unix pipes, etc.).

Some strengths of the CS connector style are portability, performance, evolvability and reusability.

Portability Client-server connectors are supported across many operating system platforms (e.g. TCP/UDP sockets, serial port communication, Unix pipes, XMLRPC, HTTP)

Performance Client-server connectors typically have the highest network performance in terms of latency and throughput because of their low-level nature.

Evolvability Because of the separation of concerns between client and server components, the design and implementation of either type of component can be iterated independently of the other provided that the interface between the client and server components within an architecture is kept consistent.

Reusability Client-server connectors contribute to the reusability of architectural elements provided clients and server implement the same interface.

Some weaknesses of CS connector styles are in configurability, scalability and robustness.

Configurability Architectures based on the client-server style are difficult to reconfigure because of the tight-coupling (in terms of space, time and synchronization) of client-server components.

Scalability As the number of nodes within a system increases, the requirement of identity of client-to-server connection or the requirement of explicit topology makes it difficult to scale up a distributed interactive system.

Robustness Also because of the explicit definition of topology, system failure can occur when a components goes offline. The likelihood of this failure mode increases as nodes are added to the system.

5.1.4.2 Remote Procedural Call

Remote procedural call (RPC) and its object oriented equivalent, remote method invocation (RMI), support communication and coordination between two or more processes by presenting an interface to execute procedures or methods on remote process that resembles the interface of locally execute procedures. Birrell et al. list the major structures of an RPC system as the client, client-stub, RPC runtime, server stub and server [BN84]. The RPC style is derived by combining client-server pairs (one for each stub-runtime pair) via a local procedure call connector with a client-server hierarchy that connects the client and server connectors within a two or more instances of the RPC runtime. An early design goal of RPC systems was to provide communication mechanisms that were as easy to use as local procedure calls to encourage the development and experimentation of distributed applications [BN84].

Some strengths of RPC-based systems are simplicity, reusability, evolvability and portability.

Simplicity RPC systems appear simple to its users because of its resemblance to local procedure calls.

Reusability RPC-based architectures inherit the reusability of the client-server style components provided their interfaces remain stable.

Evolvability RPC styles promote evolvability to the extent that the design and refinement of client and server components can be iterated independently provided their shared interfaces remain stable.

Portability The portability of RPC systems depend on the availability of cross-platform (in terms of programming languages, operating systems, etc.) ports of the underlying RPC runtime system. XML-RPC and JSONRPC are examples of widely available RPC bindings that make it possible to port RPC-based applications.

Some weakness of the RPC-style connectors are evolvability, configurability reliability.

Evolvability The RPC-style limits evolvability of a system in that the interface between client and server components is defined statically, especially for RPC implementations that provided extensive facilities for interoperability across programming languages via an interface definition language (IDL) (e.g. the Internet Communication Engine (ICE) [zer09]).

Configurability Many weaknesses of the RPC style stem from the coupling (temporal, referential and synchronization) of client and server components. This coupling makes it difficult to reconfiguration RPC-based distributed applications for large number of nodes or varying collections of nodes.

Reliability Given the design intent to present the facade of locate operation, when a remote procedure call is made, the calling process transfers control until a response can be given. This limits reliability, for instance, if the remote process hangs so does the calling process in the absence of a timeout. [1]

5.1.4.3 Tuplespaces

A tuple space is an abstract computational environment, or shared data space, within which processes add and/or remove data elements called tuples [Gel85]. Each tuple is a list of typed, named values. The basic interface of the tuple space style consists of three operations: 1. *out*, an insertion into the tuple space; 2. *in*, a tuple is copied into the calling process and removed from the tuple space; 3. and *read*, a tuple is copied but not removed from the tuple space. Gelernter describes tuples spaces as a generative model of distributed computation in that each tuple resulting from an out operation is a separate entity independent of the process

[1] Birrell et al. view timeouts as incongruous with the concept of RPC because local, synchronous procedure calls typically do not have timeouts [BN84]

that generated it. This first class-ness of data elements also enables component interaction within a tuple space-based architecture to be decoupled both in space and time [Gel85].

Within the distributed interactive space, the iROS and iStuff systems incorporate a tuple-space style to facilitate communication and coordination between nodes within the system [JFW02, BRSB03].

Some strengths of the tuplespaces as a connector are as follows:

Simplicity Tuplespace-based architectures promote simplicity in that the tuplespace abstracts the network channels over which processes may pass tuples.

Network Performance When tuplespace implementations are highly optimized and tuplespace users are shielded from the lower abstraction level details and not able to construct poorly tuned distributed applications, the tuplespace style promotes networks performance.

Scalability Tuplespace-based systems are easy to scale to large number of nodes because of the loose coupling (temporally, referentially and spatially) of interacting components.

Composability Tuplespace-based architecture promote composability because of the simple interfaces over which process coordinate and loose coupling.

Some weaknesses of the tuple space style for connectors:

Network Performance Network performance for tuplespace-based systems can difficult to determine because many tuplespace implementations utilize multiple communication mechanisms with different performance characteristics.

Scalability Tuplespace-based architectures can be constrained in their ability to scale when implementations depend on a single tuplespace instance. This limit to scalability can be addressed through the use of multiple tuplespace instances at the cost of simplicity.

Portability While the tuplespace model is language independent, in practice infrastructure for tuplespace-based computing programming language specific and so applications that are implemented in multiple programming languages cannot interoperate across tuplespace implementations.

5.1.4.4 Publish-Subscribe

The publish-subscribe style of connector enables entities to exchange data in a decoupled manner. Within this style we have two classes of components, publishers and subscribers, that communicate over a shared data bus, often called the event manager [EFGK03]. Consumers express an interest in an event or type of

event; Producers publish events and subscribers are notified of any events which match any their subscriptions registered by the system. According to Eugster, several properties of publish-subscribe-based systems result from its ability to decouple producers and consumers of information temporally, spatially and synchronously [EFGK03]. The publish-subscribe style is closely related to the event-based integration style.

Some strengths of publish-subscribe:

Evolvability Publish-subscribe-based architectures are evolvable because of the loose coupling of components. Components can be iterated independently.

Efficiency Publish-subscribe system promote efficiency for applications such as data monitoring because polling mechanisms are not needed to keep the system updated.

Extensibility It is easy to extend a system by adding new components to pub-sub systems to listen to events.

Configurability The structure and behavior of pub-sub-based system is highly configurable. For instance, spatial decoupling as enabled by publish-subscribe middleware enable one to define the behavior of producers and consumers and then compose new system topologies based on the topics specified within the collection of publishers and subscribers.

Scalability Publish-subscribe style architectures can scale easily in the number of components within the system because the interfaces for component interaction do not require identity. Also the publish-subscribe communication model matches very well with to interactions based on one-to-many or many-to-many topologies.

Some weaknesses of publish-subscribe:

Scalability Architecture based on the publish-subscribe style have limited scalability because in the simplest implementations, the event bus becomes a bottleneck for broadcasting to all subscribing components within the system. These challenges in scalability can be addressed through the use of filtering and federation of the event broadcast infrastructure, but at the cost of simplicity.

Simplicity Pub-sub style architectures are weak in the property of simplicity

Visibility Pub-sub is essentially the event-based invocation style applied to communication, and as such, uses an implicit invocation control structure scheme. Such systems are difficult to debug. One reason for this is the loose coupling of the publishers and subscribers; event producers broadcast events

without knowledge of consumers' identity. Also consumer components must specify event handlers under the assumption of non-deterministic ordering of operations.

5.2 Evaluation of the PTI Architecture

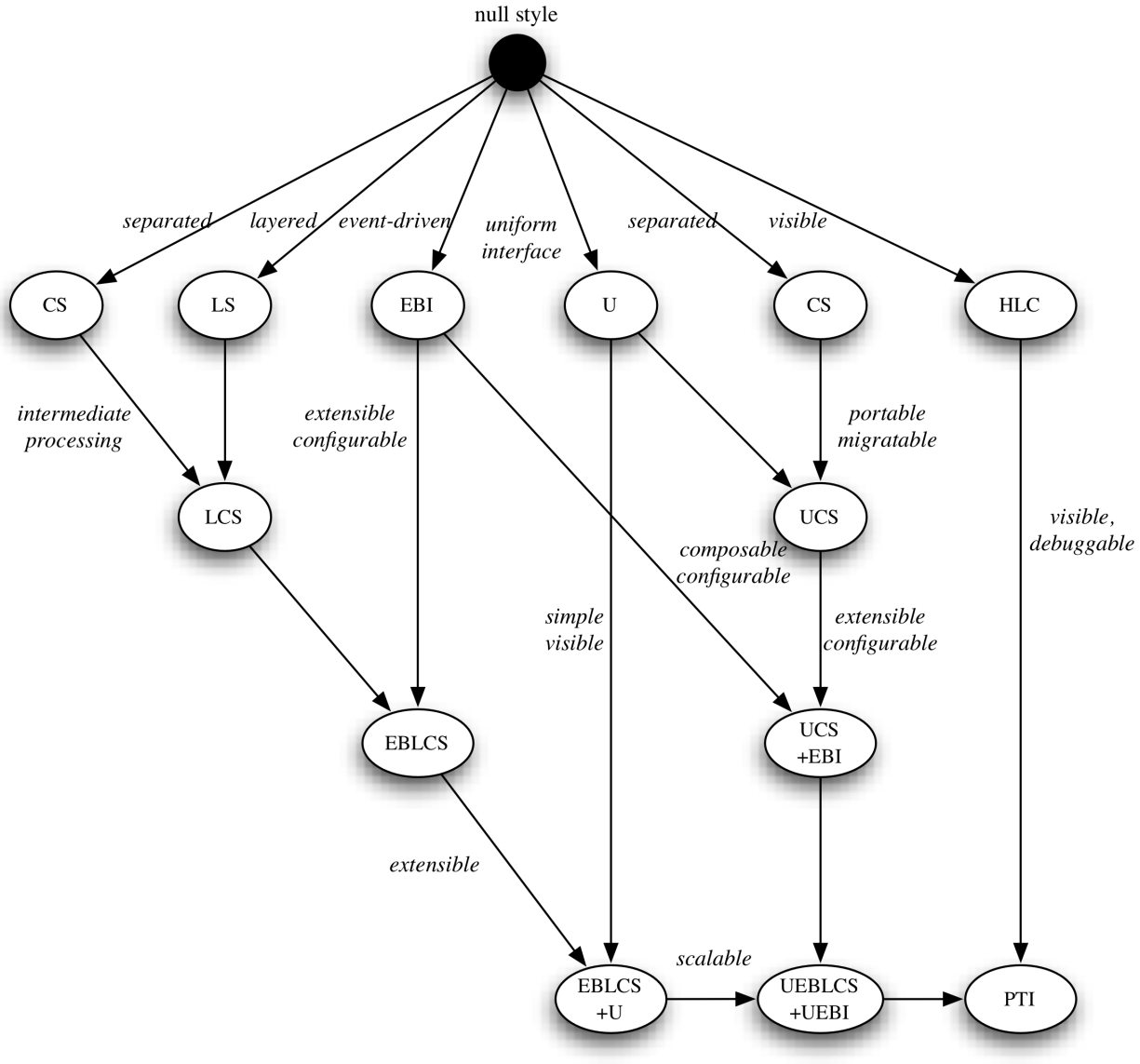

FIGURE 5.1: PTI derivation by style constraints

Now that we have surveyed various architectural styles that may be employed within a distributed, heterogeneous TUI-based system, we can evaluate the PTI architecture in terms of the architectural properties induced by its constraints. Before discussion is given to this evaluation, we will discuss the derivation of the PTI architecture from more basic styles.

5.2.1 Deriving the Proxy Tangible Interactor Style

In Figure 5.1, the derivation of the PTI architecture is illustrated in terms of desired architectural properties induced by the constraints of several architectural styles. At the top of the derivation tree, we begin with a null style in which no constraints are placed upon systems for tangible interaction.

5.2.1.1 Null Style for Distributed, Heterogeneous Tangible User Interface-based Systems

Fielding describes a null style as representing a system where there is no division between components that comprise the system [Fie00]. A null style for systems at the focus of this dissertation are somehow capable of supporting the change the state of digital systems triggered by changes in the physical realm and vice versa. In a simple case, one might implement a single program that acquires input from various local and/or remote sources and invokes the appropriate software action.

5.2.1.2 Client Server

The first style added to the PTI style is the client server architectural style. The client-server style is used within the REST architectural style on which the World Wide Web is based to separate the user interface concerns from data storage and processing concerns [Fie00]. In several interactive systems, the client-server style is applied to separate the concerns of user interaction device management from concerns of integrating the user interface and functional core [KB07a, OOJ09, UI97]. We apply this style to separate the concerns of resource management and utilization from the concerns of the user interface and the domain functional core. The term resource is used to refer to both interaction devices both for input and output, as well applications and services. By separating the user interface and functional core from, it becomes easier to redeploy application level code across platforms. This separation also supports collaboration across organizations over time who are able to iterate on various component independently.

5.2.1.3 Layered System

By constraining the behavior of the components within layers, we can simplify the design of certain components. In many cases, tangibles-based applications are developed for integration within widely deployed GUI-based system, many of which apply the layered system style. Resource adapters are one class of PTI architecture elements where the layered system style is applied. Application-level components do not see resources below the adaptation layer, but can take advantage of new resources as new adapters are written contributing to PTI's extensibility and evolution.

5.2.1.4 Event-based Integration

By applying the EBI style, PTI applications are constrained from directly accessing channels for user input and explicitly invoking procedures to handle such input. Application of this style decouples the producers and consumers of data. This style is applied at two points within the derivation of the PTI style. At one point, a TUI's behavior is defined in terms of event-driven behavior that is defined through event listeners on interactor proxies. The second application of the EBI style is on the connector interfaces between resource and application layer. This results in a publish-subscribe connector, which is discussed further in §5.2.1.6.

5.2.1.5 Uniform Interface

The interfaces between application-level proxy components and resource adapters are constrained to a uniform interface, which simplifies the interactions between components from those layers. Application of the uniform interface style also is used to meet the functional requirement to support integration of diverse implementations of a given resource provider. By constraining a resource provider (e.g. a tangible input device) to implement a limited, well defined interface, the entire architecture can be extended to support additional types of resources (i.e. extensibility, evolvability). Applying this style also promotes portability across various software platforms and improves visibility of component interaction.

5.2.1.6 Channel Agnostic Connector/Channel Agnostic Publish-Subscribe

A key design goal of the PTI architecture was to make the connection between PTI components portable across system boundaries. Achieving this would promote code reuse and system modifiability with respect to migrating from localized to distributed systems. By adding a thin interconnection layer based on an unified communication interface that abstracts both the physical channels over which components interact, the same logic for inter-component interaction across local and remote channels. This style is derived by combining the client-server, uniform interface and layered system styles.

This style is further extended to decouple (referentially, spatially and to a lesser degree temporally) PTI component interaction by constraining the delivery of messages to a publish-subscribe communication pattern. The channel agnostic publish-subscribe (CASP) is derived by combining the channel agnostic connector style with the EBI style. The addition of this style to the PTI architecture makes it easy to extend a PTI-based system through the addition of resources that publish events and subscribe to message requests. CASP also makes distributed PTI-based systems configurable because the connections between components

can be (re)defined at runtime. The publish-subscribe communication pattern also maps to higher-level communication patterns (e.g. one-to-many, many-to-one, one-to-any) and supports use cases such as the binding of a single proxy to multiple resources or vice versa. Examples of these styles can be found in systems such the ZeroMQ messaging API and various message-oriented middleware [zer10, Cur04, Mer11].

5.2.1.7 High-level Configurator

One side-effect of EBI-based systems is the diminished visibility of component interactions, especially at the code or specification level. Diminished visibility follows from the decoupled nature of the interfaces between event producing and consuming components, which were designed to be independent of the interfaces of other components. To ameliorate low visibility, higher level components can be used to provide a more global view of the distributed system. Within the PTI architecture, this role would be filled by a catalog component. Such a component would monitor the connections between PTI components, as well as the state of the resources within a PTI-based system. Application of this style adds complexity to the system; however because the EBI style is employed, such a catalog only needs to subscribe to all events, thus not adding complexity to the adapter and proxy components themselves. Effects on performance can be reduced through the use of layering, filtering and federated catalog instances. The need for a high-level interface to a distributed configuration was highlighted in real-world use within an introductory course in interface design and technology. During this experience users were not clear as to the state of resources and their bindings to running application.

5.2.2 Architectural Styles-based Assessment of PTI

In Table 5.1, we have a visualization of the non-functional architectural properties supported by the software architectures embodied by several frameworks for distributed, heterogeneous TUI-based applications. Each architecture is evaluated based on whether it promotes or limits thirteen system qualities. This is visualization is based on the evaluation technique devised by Fielding for comparing and contrasting network-base architectural styles [Fie00]. This technique is useful in that it enables one to have a high-level picture of the architectural properties of a candidate architecture without access to an implementation; however a limitation of this approach results from the fact that architectures are typically evaluated as a whole and so it is difficult to gauge the effect of a design decision in the absence of alternate implementation that corre-

TABLE 5.1: A visualization of the architectural properties of various architectures for distributed reality-based interface. This evaluation is based on Fielding's architectural style-based framework for analyzing network application architectures [Fie00]

Style	Derivation	Net Performance	UP Performance	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customizability	Configurability	Reusability	Visibility	Portability	Reliability
CS					+	+	+							
LS			-		+		+				+		+	
RR			++		+									+
UPF		-	±			++	+	+		++	++	+		
EBI				+	- -	±	+	+		+	+	-		-
TS		±				++	+			+	+	-	+	
CF						±				+		++		
iStuff	TS + UPF + EBI + CF	±-	±	+	±±			++		++	++	++		
Shared-Dictionary	RR + CS + LS + EBI		+±	+	+++	+±	+++	+		+	++	-	+	±
PTI	LS + CS + EBI + U + CF		-	±	±±	++±	+++	+		+	++	+±	+	-

Architecture Style Abbreviations
CS: client-server
LS: layered system
RR: replicated repository
UPF: uniform pipe-and-filter
EBI: event-based integration
TS: tuplespace
CF: configurator

spond with alternative architectures. Accordingly, when a design both contributes to and detracts from a architectural quality each pro and con is denoted, not combined.

5.3 Summary

This chapter presents a framework based on software architecture styles to examine the properties of the proxy tangible interactor architecture. This discussion begins by briefly overviewing several styles relevant to software architectures that support distributed, heterogeneous tangibles. These styles are applied to the architectures of several tangibles systems as described in the literature. For instance, two defining features of the software architecture of the TUIO/reacTIVision framework are based on the uniform interface and client-server architectural styles[KB07a]. Both these styles promote evolvability of the systems built upon these architectures. Several modern interactive system architecture are based upon the observer pattern, which is often applied to Mode-View-Controller or similar aggregate design patterns. Such systems are extensible, promote some degree of simplicity in adding components to a configuration and are event-driven, which maps to model of interactivity. Software architecture styles are a useful tool for analysis because the constraints imposed by a style are related to a rationale or goal of inducing some architectural property.

Given this knowledge of the architectural properties that may be induced by an architectural styles as well the architectural properties identified to support distributed, heterogeneous tangibles, we can derive an architecture to achieve those properties. We do so by iteratively applying architectural design constraints. This derivation is described in §5.2.1.

Given the styles applied to the PTI architecture, we use the architectural properties that are known to be induced by such styles to identify the architectural properties of a larger system based on the PTI architecture. We can also use this framework to contrast and compare with the architectures of related systems. Based upon this styles-based framework, PTI architecture promotes efficiency, scalability, simplicity, evolvability, extensibility, configurability, reusability, visibility and portability. Based upon its composite styles, the PTI architecture demotes user perceived performance, efficiency, scalability, simplicity, visibility and reliability. Future work would entail experiments to verify these properties and further exploration of ways to induce desired properties while mitigating negative ones.

Chapter 6

Conclusions

This research is based on the premise that distributed contexts will become the default, and as such our tools for building interactive applications should provide abstractions that reduce the incidental complexity of distributed heterogeneous computing while providing new tools for thinking about the essential complexity of distributed, heterogeneous interactive computing. Tangible user interfaces represent an intriguing interaction paradigm to explore the design space of interactive distributed, heterogeneous computing because of its pluralism and parallelism in forms and behaviors. In the theory and practice of distributed systems we find the tools and expertise needed to expand of the scope and scale of interactive environments that leverage the human knowledge and skills for engaging the physical world within the digital. However, in this intersection we find several challenges that require the expertise of stakeholders from several disciplines; and so this research seeks to provide an interface between these realms and shed some light on a path to a future in which physically engaging computing is common across physical and organizational boundaries.

In the pursuit of solutions to the challenges of distributed, heterogeneous systems for tangible interaction, this research makes the following contributions:

- Proxy Tangible Interactor (PTI), an architecture for distributed, heterogeneous systems for tangible interaction;
- and TUIKit, a software development toolkit based on the PTI architecture that enables tangible user interface-based applications to vary their configuration in terms the number, type and location of resources that comprise that application.

More specifically, the author aims to build tangible user interfaces-based systems composed of heterogeneous devices at various scales. This work began with the pursuit of TUI development tools that strike a balance between simplicity in development and the power and expressivity needed to specify potentially large scale TUI-based systems. These efforts have led to the examination of software architectures for TUIs in distributed, heterogeneous computing environments. The aim of this architectural study is to devise a system that supports extensibility and re-configurability, network performance that can support interactivity at scale, and to separate concerns of network programming from that of programming a TUI's structure and

behavior. As result of this study, I have identified an architecture that separates the logical representation of a TUI-based application from its configuration of concrete resources. I coin this architecture, the Proxy Tangible Interactor architecture. The term proxy speaks to the decoupled nature of distributed, heterogeneous interactive systems in which abstract proxy objects can be dynamically mapped to real resources. Tangible interactor speaks to an abstraction that encapsulates the high-level behavior or functionality of some element within a TUI. Combined these terms represent a progression and build on several advances in abstractions for interactive systems to address the challenges of distributed, heterogeneous tangible computing.

A tangible within the PTI architecture is represented by a proxy object that is bound to one or more concrete resources through an adapter interface. The interaction between resource proxies and adapters is fine-grained and event-based. Applications based on this architecture consists of a set of proxies representing elements of TUI-based application (e.g. tangible input and output devices, software applications and services, etc.) which are loosely coupled with a set of concrete resources. In other words, the Proxy Tangible Interactor architecture is based on the concept of loose coupling (spatial and semantic) between abstract representations of a TUI and the concrete resource on which these resources execute.

TUIKit is a software toolkit for building TUI-based applications in distributed, heterogeneous computing environments. TUIKit both inspired the design of the PTI architecture and is reciprocally influenced by the insights of the PTI design process. With TUIKit, developers are able to vary the number, type and locality of tangible interaction devices that comprise a distributed TUI-based application with minimal changes to code. Furthermore, many of these TUI-base application parameters are reduced from properties to be changed at the source code level to properties that can be configured (and reconfigured) at run-time. This capability makes systems built with TUIKit more flexible and easier to adapt for various use contexts. This flexibility also opens a pathway for TUI-based systems to be augmented with intelligence and automation to become context-aware [Dey01].

Following the PTI architecture, TUIKit-based applications have a layered system architecture. Application level components interact with resource layer components through a light-weight messaging and adaptation layer. The application layer consists of a set of proxies that expose an interface based on the tangible interactor abstraction with which developers can define the structure and behavior of TUI. The resource layer consists of a set of adapters that mediate communication with a resource by translating between the implementation specific interface for a particular resource and a more general interface for class

of resources to which the target resource belongs. This classification can be based on behavior (e.g. the class of dial input controls) or functionality (e.g. the class of sensors that detect identity such as a visual tag system or a RFID module). The messaging layer through which proxies and adapters interact abstracts the physical channels over which this communication might occur. This combination of semantic and spatial loose coupling enables a large degree of code reuse in systems with a high degree of variability.

6.1 Dissertation Research Implications

The technical implications of this research are improved component reuse, interoperability and portability. From these system qualities follow several implications.

Interoperability Across Tangible User Interface Platforms Interoperability is defined as

> the ability of two or more systems or components to exchange information and to use the information that has been exchanged.[GKM+91]

This system quality is related to but is distinguished from portability, which is the

> ease with which a system component can be transferred from one hardware or software environment to another. [GKM+91]

TUIKit's portability is partly evidenced by the existence of several language bindings mostly written by this dissertation's author. TUIKit and its underlying PTI architecture supports interoperability of system components across operating systems, programming languages and potentially application-level frameworks as evidenced by fact that several applications have been composed of adapters written in python running on a Linux-based system while the application-level proxies run in application deployed on Windows, Mac OS X and Linux environments. These properties can be exploited achieve interoperability of TUI-based application components across tangible frameworks. For instance, with the development of a TUIO tracker adapter, TUIKit applications that incorporate spatial interactors can be mapped on TUIO-based trackers. Alternately, with the development of components that adapt TUIKit-based devices into the TUIO tracker interface, TUIO code can run atop TUIKit-based interaction devices.

Promotion of the Sharing of Tangible Computing Applications Another key technical property of the PTI architecture and TUIKit is the promotion of code or component reuse. Component reusability is pro-

moted in part by the separation of the code that defines the tangible user interface and domain logic from the code that defines lower-level logic for sensing and actuation. The field of tangible user interfaces abounds in diverse hardware and software technologies for sensing and actuation. In the high likelihood that interaction platforms are not common across groups, they can still share application-level code with a greater likelihood of interoperability and increased portability as supported by architectural approaches such as PTI.

More Tools for Evaluating Tangible Computing Applications Following from the potential increase in sharing of tangible computing applications, TUI developers and researches have more tools for evaluating tangible computing applications. Students and researchers can more easily re-implement systems shared within the TUI community and run them against whatever hardware they have available. To the extent that diverse tangible interaction hardware technologies possess the same properties (e.g. affordance, etc.) as the original resources against which a tangible application is developed, any assertions made about the application can be tested and either confirmed or contradicted.

More Tools for Evaluating Tangible Interface Technologies Correspondingly, greater reuse of code tangible computing applications supports the evaluation of sensing and actuation technologies for TUIs. For instance, tangible computing application be used as a benchmark against which diverse tangible interaction platform resources can be evaluated. The PTI architecture's support of extensibility in the support of TUI hardware means that TUI elements can be evaluated both in isolation and as part of a larger system. A TUI technologist can focus on new interaction modalities and have an ecosystem of applications and complimentary interaction device with which to integrate.

Facilitates Collaboration Between Stakeholders in the Distributed, Heterogeneous Tangible Interaction Design Space A fundamental property of software architecture is that it defines the major structures of a complex software-intensive system as well the interfaces between that system's strutters. The modularity of the PTI architecture and its initial implementation, TUIKit, define requirements and scope of the various stakeholders enumerated in §1. Armed with the information concerning the interfaces of components within the PTI architecture, various components can be developed independently with a reasonable expectation of integratibility. This has already been alluded to as it relates to the interplay of the evolution of the software and hardware components of a TUI-based system. But the PTI architecture supports

the decoupling of a TUI's structure and behavior from its configuration as a collection of components that pass messages over a number of network connections. The details of networking are mostly hidden from a TUIKit when developing a distributed TUI-based application. One implication of this separation, is that as the infrastructure for communication and coordination of resources within a distributed, heterogeneous tangible interaction system is improved, applications built upon this infrastructure can reap the benefit of these improvements. These are just a few examples of the implications of this architectures and corresponding systems toward facilitating collaboration among distributed, heterogeneous stakeholders.

6.2 Future Work

Usability Studies for the PTI architecture and TUIKit Some ultimate goals of the efforts described in this dissertation are to make it easier to developed TUI-based applications in distributed, hererogeneous compting environments as well enabling developers to build larger scale, more complex systems for distributed tangible interaction. In the pursuit of these goals, a software architecture or development toolkit can be viewed as a user interfaces. Future work along this line of research might involve usability studies of artifacts of the PTI or derivative architectures as well TUIKit. System tools that both have a low threshold and well as high ceiling are often design goals of many frameworks for building interactive systems [Mye95]. Research such as the cognitive dimensions of notation and API usability might speaks to methodology for evaluating usability of the PTI architecture and TUIKit framework.

Performance Analysis of the PTI Architecture and TUIKit Several research questions center around the performance and scalability that is supported by the PTI architecture and TUIKit-based applications. Ultimately, the architecture supported by PTI depends on the level of refinement and specificity of its architecture. For instance, to the extent that the process architecture and corresponding physical architecture were defined at the level of a protocol, then it would be possible to formulate a performance model for PTI-based systems. If PTI architecture was defined at the level of an implemented system in execution, then actual metrics could be taken for a given system load. Accordingly, the performance of TUIKit and thus the PTI architecture have only been measured for small systems. Future work would entail the more extensive tests of the performance and scalability characteristics of the PTI architecture and TUIKit framework.

Refinement of Use Cases for Distributed, Heterogeneous Tangible Interaction One of the more challenging aspects of this research has been devising use cases that are rooted in real world problems, yet are illustrative of frontiers of we cannot yet reach within the tangible user interface design space. Use cases are an important tool in communicating the value of distributed, heterogeneous computing to the field of TUI research and development, as well as highlighting problems(technical, social or otherwise) to be addressed in making building TUIs in distributed, heterogeneous environments not only feasible, but effective in addressing problems for which tangible interaction is relevant. Further research of architectures for distributed, heterogeneous TUIs would benefit greatly from use cases that push the technical limits of what is presently possible and concisely connect the various aspects (e.g. conceptual, logical, systematic, physical, etc.) of such systems.

Algorithms, Models and Engineering Practices for Scalability and Network Performance in Distributed Interactive Systems There are many open problems for systems that support TUIs in distributed, heterogeneous environments. For instance, latency cannot totally be eliminated from network-bound operations within a distributed tangible interactive system. So the questions becomes, what strategies can be employed to mitigate the effects of latency within distributed interactive systems? Future research of software architectures might involve an study of techniques such latency hiding. System designers would also benefit from knowledge as it relates to certain use cases for which distributed system are not a workable solution. Perhaps this state is contextual (e.g. for a given networked resource load, a system cannot support performance for remote distributed interactivity at scale.) Perhaps these challenges can be addressed through system engineering techniques such as federation to address performance at scale. Future research is also needed to address models that related properties of interactive experience to architectural qualities of systems that support interactive contexts.

6.3 Closing Remarks

An inspiration for this research was systems such as Open Graphics Library (OpenGL) [Khr12b] standard and the emerging Open Computing Language (OpenCL) [Khr12a]. Many graphics programmers are afforded the luxury of being unaware of how much parallelism is inherent in the hardware designs of graphics processing units (GPUs). With respect to graphics programming, OpenGL provides APIs based on a

high-level model of a graphics pipeline and if the GPU has parallel processing hardware, its OpenGL implementation is responsible for parallelizing the code that can exploit it. Furthermore, OpenCL generalizes several efforts to exploit the hardware parallelism of GPUs for general purpose by providing a uniform interface for concurrent programming that run on heterogeneous parallel processors (i.e. GPU, central processing units, etc.). Also within the graphical computing space, we find efforts such as Processing [Pro12] that provide further abstractions on top of graphics systems like OpenGL and additional tools to enable its users to ignore certain technical aspects of programming and focus on the aspects of visual computing relevant to produce an desired output. Such systems provide a low barrier to entry to produce visuals and interactivity, but also ease the path into more technically complex visual and interactive applications.

A hope for the PTI architecture and TUIKit is the development of a system model for distributed, heterogeneous tangible computation. Parallel to this system model of distributed, heterogeneous tangible computation is an abstract model of a TUI-based application. This model would define a kernel that encapsulates the high-level structure and behavior of a TUI or TUI element. To manage the complexity of the many components that may comprise a distributed, heterogeneous tangible, algorithms and services would aid the user in mapping that high-level kernel of tangible computation onto distributed, heterogeneous resource for tangible interaction as available within an application's environment. Inspired by this vision, I set out to define an architecture in which components could easily be (re)connected to compose different system configurations. This was the rationale for adopting a publish-subscribe-based connector as a substrate for system integration. The development and release of ZeroMQ was fortuitous, but hoped for since the author's master's thesis research [TJM05] [1]. With the ability to separate communication logic from the physical channels of communication, that logic could be reused in various distributed system configurations that use the optimal connector for a given context (e.g. local components communicate over intra-process or inter-process channels whereas remote components communicate over network channels). To address integration of heterogeneous resources, components would interact via a uniform interface, which was the rationale for extending the tangible interactor abstraction to distributed environments. As long as a resource conformed with an interface of an interactor proxy belonging to some TUIKit-based application context, that resource could be bound to that proxy.

[1] This research was concerned with ways to composed distributed visualization applications over a connection interface that mapped to intra-process, inter-process, intra-node and inter-node communication boundaries.

By combining thee above architectural features, applications based on the PTI architecture could vary in the number, type and location of resources that comprise a distributed, heterogeneous tangible user interface-based system. This capability of the PTI architecture is demonstrated by several examples built with TUIKit, a toolkit for developing distributed, heterogeneous TUI-based applications. It is my hope that this dissertation research represents a significant step toward the system model and supporting architecture envisioned at the beginning of the closing remarks. It also my hope the architecture presented in this thesis is useful in the realization of the large scale, physically engaging computational systems.

Bibliography

[AAG95] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Softw. Eng. Methodol.*, 4:319–364, October 1995.

[ADG+05] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. v. Nieuwpoort, A. Reinefeld, F. Schintke, T. Schuett, E. Seidel, and B. Ullmer. The grid application toolkit: Toward generic and easy application programming interfaces for the grid. *Proc. of the IEEE*, 93(5):534–550, 2005.

[Ais79] R. Aish. 3D input for CAAD systems. *Computer-Aided Design*, 11:66–70, March 1979.

[And91] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Comput. Surv.*, 23(1):49–90, March 1991.

[BAC+08] Leonardo Bonanni, Jason Alonso, Neil Chao, Greg Vargas, and Hiroshi Ishii. Handsaw: tangible exploration of volumetric data by direct cut-plane projection. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 251–254, New York, NY, USA, 2008. ACM.

[BBE+02] Victoria Bellotti, Maribeth Back, W. Keith Edwards, Rebecca E. Grinter, Austin Henderson, and Cristina Lopes. Making sense of sensing systems: five questions for designers and researchers. In *Proc. of CHI'02*, pages 415–422, 2002.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[Bde09] Ayah Bdeir. Electronics as material: littlebits. In *TEI '09: Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, pages 397–400, New York, NY, USA, 2009. ACM.

[BID98] Scott Brave, Hiroshi Ishii, and Andrew Dahley. Tangible interfaces for remote collaboration and communication. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 169–178, New York, NY, USA, 1998. ACM.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[BRSB03] Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. istuff: a physical user interface toolkit for ubiquitous computing environments. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 537–544, New York, NY, USA, 2003. ACM.

[BSF04] R. Ballagas, A. Szybalski, and A. Fox. Patch panel: Enabling control-flow interoperability in ubicomp environments. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 241–252, 2004.

[BYV+09] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[Che83] P.P.S. Chen. English sentence structure and entity-relationship diagrams. *Information Sciences*, 29(2-3):127–149, 1983.

[CRR08] Nadine Couture, Guillaume Rivière, and Patrick Reuter. Geotui: a tangible user interface for geoscience. In *TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages 89–96, New York, NY, USA, 2008. ACM.

[Cur04] E. Curry. Message-oriented middleware. *Middleware for communications*, pages 1–28, 2004.

[CvdB06] S. Ciraci and PM van den Broek. Evolvability as a quality attribute of software architectures. In *The International ERCIM Workshop on Software Evolution 2006, LIFL et l'INRIA, Universite des Sciences et Technologies de Lille, France*, pages 29–31. UMH, 2006.

[Deu01] P. Deutsch. The eight fallacies of distributed computing. *Larsen, JE*, 2001.

[Dey01] A.K. Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[DG97] P. Deutsch and J. Gosling. The Eight Fallacies of Distributed Computing, 1997.

[Dix08] A. Dix. *Network-based Interaction in The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*. Second edition, 2008.

[DLG+08] Bruno Dumas, Denis Lalanne, Dominique Guinard, Reto Koenig, and Rolf Ingold. Strengths and weaknesses of software architectures for the rapid creation of tangible and multimodal interfaces. In *TEI '08: Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages 47–54, New York, NY, USA, 2008. ACM.

[Dou04] P. Dourish. *Where the action is*. MIT press Cambridge, MA, 2004.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[ENP10] W. Keith Edwards, Mark W. Newman, and Erika Shehan Poole. The infrastructure problem in hci. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 423–432, New York, NY, USA, 2010. ACM.

[FFF80] J.H. Frazer, J.M. Frazer, and P.A. Frazer. Intelligent physical three-dimensional modelling system. *Proc. of Computer Graphics*, 80:359–370, 1980.

[FIB95] George Fitzmaurice, Hiroshi Ishii, and William Buxton. Bricks: Laying the Foundations for Graspable User Interfaces. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, 1995.

[Fie00] R.T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[Fie05] L. Fiege. *Visibility in event based systems*. PhD thesis, Universitäts-und Landesbibliothek Darmstadt, 2005.

[Fis04] K.P. Fishkin. A taxonomy for and analysis of tangible interfaces. *Personal and Ubiquitous Computing*, 8(5):347–358, 2004.

[Fit96] G.W. Fitzmaurice. *GRASPABLE USER INTERFACES*. PhD thesis, University of Toronto, 1996.

[FKNT02] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, volume 22, page 2002. Citeseer, 2002.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200, 2001.

[GAACB95] C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm. On the definition of software system architecture. In *Proceedings of the First International Workshop on Architectures for Software Systems*, pages 85–94. Seattle, Wa, 1995.

[GB02] Saul Greenberg and Micheal Boyle. Customizable Physical Interfaces for Interacting with Conventional Applications. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, 2002.

[GC03] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[Gel85] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[GF01] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218. ACM New York, NY, USA, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley Reading, MA, 1995.

[GIM+04] C. Greenhalgh, S. Izadi, J. Mathrick, J. Humble, and I. Taylor. Ect: a toolkit to support rapid construction of ubicomp environments. In *Proceedings of UbiComp*, volume 4, 2004.

[GJK+06] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-level application programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[GKM+91] A. Geraci, F. Katki, L. McMonegal, B. Meyer, J. Lane, P. Wilson, J. Radatz, M. Yee, H. Porteous, and F. Springsteel. Ieee standard computer dictionary: Compilation of ieee standard computer glossaries. 1991.

[Gli07] M. Glinz. On non-functional requirements. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*, pages 21–26. IEEE, 2007.

[Gof03] M. Goff. *Network distributed computing: fitscapes and fallacies*. Prentice Hall Professional Technical Reference, 2003.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[HAB+06] A. Hutanu, G. Allen, S. Beck, P. Holub, H. Kaiser, A. Kulshrestha, M. Liska, J. Jon MacLaren, L. Matyska, R. Paruchuri, S. Prohaska, E. Seidel, and B. Ullmer. Distributed and collaborative visualization of large datasets using high-speed networks. *Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications*, 22(8):1004–1010, 2006.

[HAG+09] A. Hutanu, G. Allen, J. Ge, C. Toole, E. Schnetter, P. Diener, E. Bentivenga, R. Parachuri, and K. Liu. Large scale problem solving using automatic code generation and distributed visualization. Second IEEE International Scalable Computing Challenge (SCALE 2009), 2009.

[HB06] E. Hornecker and J. Buur. Getting a grip on tangible interaction: a framework on physical space and social interaction. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, page 446. ACM, 2006.

[Hol09] P. Holleis. Programming Interactive Physical Prototypes. In *Proc. 1st Int'l Workshop on Design and Integration Principles for Smart Objects*, 2009.

[HPGK94] K. Hinckley, R. Pausch, J.C. Goble, and N.F. Kassell. Passive real-world interface props for neurosurgical visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, pages 452–458. ACM New York, NY, USA, 1994.

[HYA+08] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 91–100, New York, NY, USA, 2008. ACM.

[IEE00] IEEE Standards Association. 1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. `http://standards.ieee.org/findstds/standard/1471-2000.html`, October 2000.

[Ish08] H. Ishii. Tangible bits: beyond pixels. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*. ACM New York, NY, USA, 2008.

[IU97] Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241, New York, NY, USA, 1997. ACM.

[IU10] Hiroshi Ishii and Brygg Ullmer. *Tangible User Interfaces*. Fourth edition, 2010.

[JF02] B. Johanson and A. Fox. The event heap: A coordination infrastructure for interactive workspaces. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 83–93, 2002.

[JF04] B. Johanson and A. Fox. Extending tuplespaces for coordination in interactive workspaces. *Journal of Systems and Software*, 69(3):266, 2004.

[JFW02] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *Pervasive Computing, IEEE*, 1(2):67–74, 2002.

[JGH+08] Robert J.K. Jacob, Audrey Girouard, Leanne M. Hirshfield, Michael S. Horn, Orit Shaer, Erin Treacy Solovey, and Jamie Zigelbaum. Reality-based interaction: a framework for post-wimp interfaces. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 201–210, New York, NY, USA, 2008. ACM.

[JKGB05] S. Jorda, M. Kaltenbrunner, G. Geiger, and R. Bencina. The reactable*. In *Proceedings of the International Computer Music Conference (ICMC 2005), Barcelona, Spain*, pages 579–582. Citeseer, 2005.

[Jor10] Sergi Jordà. The reactable: tangible and tabletop music performance. In *CHI EA '10: Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*, pages 2989–2994, New York, NY, USA, 2010. ACM.

[KB07a] Martin Kaltenbrunner and Ross Bencina. reactivision: a computer-vision framework for table-based tangible interaction. In *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 69–74, New York, NY, USA, 2007. ACM.

[KB07b] Martin Kaltenbrunner and Ross Bencina. reactivision: a computer-vision framework for table-based tangible interaction. In *TEI '07*, pages 69–74, New York, NY, USA, 2007. ACM.

[KBBC05] M. Kaltenbrunner, T. Bovermann, R. Bencina, and E. Costanza. TUIO: A protocol for table-top tangible user interfaces. In *Proc. of the The 6th Int'l Workshop on Gesture in Human-Computer Interaction and Simulation*. Citeseer, 2005.

[KF02] T. Kindberg and A. Fox. System software for ubiquitous computing. *Pervasive Computing, IEEE*, 1(1):70 – 81, jan-mar 2002.

[Khr12a] Khronos Group. OpenCL – The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`, 2012.

[Khr12b] Khronos Group. OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org`, 2012.

[KHT06] Scott R. Klemmer, Björn Hartmann, and Leila Takayama. How bodies matter: five themes for interaction design. In *DIS '06: Proceedings of the 6th conference on Designing Interactive systems*, pages 140–149, New York, NY, USA, 2006. ACM.

[KKVL05] D.F. Keefe, D.B. Karelitz, E.L. Vote, and D.H. Laidlaw. Artistic collaboration in designing VR visualizations. *IEEE Computer Graphics and Applications*, pages 18–23, 2005.

[KL09] S.R. Klemmer and J.A. Landay. Toolkit Support for Integrating Physical and Digital Interactions. *Human-Computer Interaction*, 24(3):315–366, 2009.

[KLLL04] Scott R. Klemmer, Jack Li, James Lin, and James A. Landay. Papier-mache: toolkit support for tangible input. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 399–406, New York, NY, USA, 2004. ACM.

[KOS06] P. Kruchten, H. Obbink, and J. Stafford. The past, present, and future for software architecture. *Software, IEEE*, 23(2):22–30, 2006.

[Kru95] P.B. Kructhen. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[KTK+05] N. Kobayashi, E. Tokunaga, H. Kimura, Y. Hirakawa, M. Ayabe, and T. Nakajima. An input widget framework for multi-modal and multi-device environments. In *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 63–70. IEEE Computer Society, 2005.

[Kum09] M. Kumpf. Trackmate: Large-scale accessibility of tangible user interfaces. Master's thesis, MIT, 2009.

[KWWW94] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994.

[Mar08] N. Marquardt. Developer Toolkit and Utilities for Rapidly Prototyping Distributed Physical User Interfaces. Master's thesis, Bauhaus University, 2008.

[Mer11] A. Merzky. Saga api extension: Message api. `http://www.gridforum.org/documents/GFD.178.pdf`, March 2011.

[MG07] Nicolai Marquardt and Saul Greenberg. Distributed physical interfaces with shared phidgets. In *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 13–20, New York, NY, USA, 2007. ACM.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[MQA+04] Jean-Bernard Martens, Wen Qi, Dima Aliakseyeu, Arjan J. F. Kok, and Robert van Liere. Experiencing 3d interactions in virtual reality and augmented reality. In *EUSAI '04: Proceedings of the 2nd European Union symposium on Ambient intelligence*, pages 25–28, New York, NY, USA, 2004. ACM.

[MvdH09] A. Mazalek and E. van den Hoven. Framing tangible interaction frameworks. *AI EDAM*, 23(03):225–235, 2009.

[Mye90] Brad A. Myers. A new model for handling input. *ACM Trans. Inf. Syst.*, 8(3):289–320, 1990.

[Mye95] Brad A. Myers. User interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 2(1):64–103, 1995.

[OOJ09] D. Olsen and D.R. Olsen Jr. *Building interactive systems: principles for human-computer interaction*. Course Technology Ptr, 2009.

[Per76] R. Perlman. Using computer technology to provide a creative learning environment for preschool children. *MIT Lego Memo*, #24, 1976.

[PNM+04] Ivan Poupyrev, Tatsushi Nashida, Shigeaki Maruyama, Jun Rekimoto, and Yasufumi Yamaji. Lumen: interactive visual and shape display for calm computing. In *ACM SIGGRAPH 2004 Emerging technologies*, SIGGRAPH '04, pages 17–, New York, NY, USA, 2004. ACM.

[Pro12] Processing.org. processing.org. `http://processing.org/` Last accessed on May 17, 2012., 2012.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17:40–52, October 1992.

[QM05] W. Qi and J.B. Martens. Tangible user interfaces for 3D clipping plane interaction with volumetric data: a case study. In *Proceedings of the 7th international conference on Multimodal interfaces*, pages 252–258. ACM New York, NY, USA, 2005.

[QMvLK05] W. Qi, J.B. Martens, R. van Liere, and A. Kok. Reach the virtual environment: 3D tangible interaction with scientific data. In *Proceedings of the 17th Australia conference on Computer-Human Interaction: Citizens Online: Considerations for Today and the Future*, pages 1–10. Computer-Human Interaction Special Interest Group (CHISIG) of Australia Narrabundah, Australia, Australia, 2005.

[rea12] reactable.com. Reactable - products. `http://www.reactable.com/products/`, 2012.

[RGO06] A. Rotem-Gal-Oz. Fallacies of distributed computing explained. `http://www.rgoarchitects.com/Files/fallacies.pdf`, 2006.

[RJR85] S.T. Redwine Jr and W.E. Riddle. Software technology maturation. In *Proceedings of the 8th international conference on Software engineering*, pages 189–200. IEEE Computer Society Press, 1985.

[RWP+04] C. Ratti, Y. Wang, B. Piper, H. Ishii, and A. Biderman. PHOXEL-SPACE: an interface for exploring volumetric data with physical voxels. In *Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques*, pages 289–296. ACM New York, NY, USA, 2004.

[SC97] M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Styles. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 6–13, 1997.

[SH10] O. Shaer and E. Hornecker. *Tangible User Interfaces*, volume 3 of *Foundations and Trends in Human-Computer Interaction*. Now Publishers, 2010.

[sif12] sifteo.com. Sifteo - sifteo cubes. `https://www.sifteo.com/product`, 2012.

[Sin92] Alok Sinha. Client-server computing. *Commun. ACM*, 35:77–98, July 1992.

[SJ09] Orit Shaer and Robert J.K. Jacob. A specification paradigm for the design and implementation of tangible user interfaces. *ACM Trans. Comput.-Hum. Interact.*, 16:20:1–20:39, November 2009.

[SK95] Hideyuki Suzuki and Hiroshi Kato. Interaction-level support for collaborative learning: Algoblock—an open programming language. In *CSCL '95: The first international conference on Computer support for collaborative learning*, pages 349–355, Hillsdale, NJ, USA, 1995. L. Erlbaum Associates Inc.

[spi12] spinmaster.com. AppMATes are the First Physical Toys That Digitally Interacts and Magically Come to Life on an iPAD! `http://www.spinmaster.com/pr/spring_2011/pdf/AppMates_2011_Product_Description.pdf`, 2012.

[Str11] Norbert A. Streitz. Tangible, Embedded, Embodied Interaction: What's the Fuss? Closing Keynote Panel at TEI'11, January 2011.

[Sub04] Sriram Subramanian. *Tangible Interfaces for Volume Navigation*. PhD thesis, Technische Universiteit Eindhoven, 2004.

[SUR+09] Rajesh Sankaran, Brygg Ullmer, Jagannathan Ramanujam, Karun Kallakuri, Srikanth Jandhyala, Cornelius Toole, and Christopher Laan. Decoupling interaction hardware design using libraries of reusable electronics. In *TEI '09: Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, pages 331–337, New York, NY, USA, 2009. ACM.

[SWK+04] E. Sharlin, B. Watson, Y. Kitamura, F. Kishino, and Y. Itoh. On tangible user interfaces, humans and spatiality. *Personal and Ubiquitous Computing*, 8(5):338–346, 2004.

[TJM05] C. Toole Jr and L. Moore. Brokering Grid Services for Distributed Visualization. In *Proceedings of the 2005 Compframe Workshop*, 2005.

[TMA+96] R.N. Taylor, N. Medvidovic, K.M. Anderson, Jr. Whitehead, E.J., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390 –406, jun 1996.

[tui11] tuio.org. Software Impementing TUIO. `http://tuio.org/?software`, 2011.

[UHBH03] B. Ullmer, A. Hutanu, W. Benger, and H.C. Hege. Emerging tangible interfaces for facilitating collaborative immersive visualizations. *computing*, 2, 2003.

[UI97] B. Ullmer and H. Ishii. The metaDESK: models and prototypes for tangible user interfaces. In *Proc. of UIST'97*, pages 223–232, 1997.

[UI99] J. Underkoffler and H. Ishii. Urp: a luminous-tangible workbench for urban planning and design. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, page 393. ACM, 1999.

[UI00] B. Ullmer and H. Ishii. Emerging frameworks for tangible user interfaces. *IBM systems journal*, 39(3):915–931, 2000.

[UIJ03] B. Ullmer, H. Ishii, and R.J.K. Jacob. Tangible query interfaces: Physically constrained tokens for manipulating database queries. *Human-Computer Interaction*, page 279, 2003.

[UJRT07] B. Ullmer, S. Jandhyala, Sankaran R., and C. Toole. Cyber infrastructure for k-12 education: A demonstration. National Workshop on Stimulating and Sustaining Excitement and Discovery in K-12 STEM Education, July 2007.

[VG07] Nicolas Villar and Hans Gellersen. A malleable control structure for softwired user interfaces. In *TEI '07: Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 49–56, New York, NY, USA, 2007. ACM.

[WB96] Mark Weiser and John Seely Brown. Designing calm technology. *PowerGrid Journal*, 1, 1996.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific America*, 1991.

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36:75–84, July 1993.

[WF97] M. Wright and A. Freed. Open sound control: A new protocol for communicating with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, pages 101–104, 1997.

[zer09] ZeroC – The Internet Communications Engine(ICE). `http://zeroc.com/ice.html`, 2009.

[zer10] Zero Message Queuing Protocol. `http://zeromq.org`, 2010.

[Zim80] H. Zimmerman. The ISO Model of Architecture for Open System Interconnection. *IEEE Transactions on Communication*, pages 425–432, 1980.

Appendix A

The TUIKit Application Programming Interface

A.1 Introduction

TUIKit is a library for integrating distributed heterogeneous resources for tangible interaction. These include physical controls, computer vision systems, electronic-mechanical devices, touch-screens and so on. The current version includes components for adapting input from various input devices, an API for event messages, an API for generic abstract interaction components and components for mediating between users/programmers and resources for tangibles-based application

A.2 TUIKit.Message

One of the key elements within the TUIKit architecture is a message. There are currently two types of messages defined witihin architecture: event notifications and requests.

A.3 TUIKit.Event

This object encapsulates user interaction event message.

A.3.1 Data Members

`string handle` URI-like address or path to concrete resource on which event originated

`TUIKit.Interactor source` reference to Interactor object that triggers the event in user space

`TUIKit.EventTypes event_type` type of event

`object state` the value of the interaction device at the time the event is generated

`int ID` unique identifier for an event, should be based on some ordering scheme

`int time_stamp` system time at the point of the events generation

`float clock_offset` offset of system clock from some global clock (e.g. NTP)

A.3.2 Methods

- `string serialize(TUIKitEvent event)`
- `TUIKit.Event deserialize(string json_encoded_str)`

A.3.3 TUIKit.Events.Types

- ROTATE,
- BUTTONDOWN,
- BUTTONUP,
- TAGENTRY,
- TAGEXIT,
- SLIDE,
- POINT,
- DEFAULT

A.3.4 TUIKit.EventListener

An interface for objects that implement event-driven behavior for a particular event on a particular Interactor

A.3.5 TUIKit.Command

TBD - So far TUIKit supports integration of tangible input devices via user interaction event messages. It probably useful to support additional message types for communication with various types of resources. Command messages are envisioned for communication with services and components which provide output.

A.4 TUIKit.Interactor

Interactors are proxies for components that comprise a tangibles-based application.

A.4.1 Data Members

`object state` This object contains the current value of parameter(s) that may be embedded within a given Interactor. Some Interactors may be stateless, so this value may be a delta.

A.4.2 Methods

`void addListener( TUIKit.EventType type, TUIKit.EventListener listener)`
This methods adds an object that implements the EventListener interface for handling events of the specified type.

`void removeListener(TUIKit.EventListener listener)` This methods removes the listener object from the collection of listeners for the Interactor object receiving the removeListener call. In the python implementation this method apparently removes the listener on all events for which it has been registered. This may not be desired.

`void removeListener(TUIKit.EventType type, TUIKit.EventListener)` This method only removes a listener for a given event type. This could replace the other version and provide the original behavior if the user pass in a null/nil or some other special value.

`void triggerEvent(TUIKit.Event event)` This method invokes the event parameter on the receiving Interactor, which in turns triggers a notification to all listeners on that event type.

A.5 TUIKit.Controller

This object communicates on behalf of and maps TUIKit.Event(s) onto TUIKit.Interactor objects.

A.5.1 Methods

`void addDevice(string handle)` This method adds an Interactor object associated with the handle parameter.

`void removeDevice(string handle)` This method removes the Interactor associated with the pided handle.

`TUIKit.Interactor getInteractor(string handle)` This method returns a reference to the Interactor object associated with the provided handle.

`void start()` This method starts the indefinite loop for collecting and dispatching events and commands.

`void quit()` This methods stops event collection routines.

Appendix B

An Interactive-Exa-Scale Visualization Future

B.1 Introduction

The following is a hypothetical scenario intended to illustrate a use case for large-scale interactive computational science and distributed visualization. An organization operates a scientific instrument that will generate in its advanced configuration an exabyte of data per run. In addition to the data from experiments, simulations for the purpose of verification and theory formulation generate hundreds of petabytes of output data per run. In many cases these volumes of data are too large to be stored in full fidelity and if stored at all too large to move around for later analysis.

Up until this point, the system architects of these facilities have called for the deployment of high-end resources for analysis, including graphics clusters and massively parallel file clusters. For visual analysis, promising HPC architecture designs often employ high ratios of graphics processors to application processors along with strategies to lessen bandwidths bottlenecks along the graphics pipeline. Despite these advances in system architectures that scale up to tackle exascale computational challenges, many organizations also employ architectures to support applications that scale out over many systems instances.

In such future we would have the ability to wield millions of cores, many exabytes of storage, billions of pixels in graphics capability and many terabits per second in aggregate network bandwidth spread across hundreds of HPC systems deployed all over the globe; but how individuals wield this power? What interfaces would enable perhaps hundreds of users to collaboratively visualize this data? How might interactive systems enable users to observe simulation in situ and do live steering of the computation? How might computational scientists be empowered to debug and refine their algorithms informed visual analysis of scientific codes in execution? The following text describes computational science use cases that are not currently possible or feasible but might be in the future given systems with interactive architectures of the sort at the focus of my dissertation research.

B.2 Live Visualization Scenario

The Big Machine is spewing out gobs of data, about an Exabyte per day. Several visualization clusters deployed at international computational science research centers have been chosen to be part of the live visualization corps. The Big Machine center has 100 GB/s in network bandwidth and each visual analysis corps system has 10 Gb/s links to the Big Machine center. At each viz corps site are on the order of 10 scientist and visualization analysts. The viz corps personnel have various individual interactive terminals as well a large shared ultra-high-definition display.

During this live analysis session, viz corps sites are each streamed a sub-domain of the data. Visualization images are rendered in real-time and streamed to other sites for compositing of the total visualization, which is displayed at each visualization site. Several strategies for parametric control can be employed, including one extreme in which each viz site has only control of their data domain and the other in which one site can drive the parameter exploration at all sites within the live session. Within a viz corps site, collaborative control can be further distributed or centralized amongst the site participants. Regarding the earlier mentioned interactive terminals, some terminals may be purely graphical and provided by laptops, tablets and smaller personal mobile devices; while other interactive terminals may incorporate physical and environmental media for engagement.

Consider this scenario. At visualization corps site Alpha, Dr. Leslie is observing the live visualization feed. She recalls earlier analysis of data from an earlier Big Machine run with the same codes. Each viz corps site has the capability to cache about 100 timesteps. She would like to apply the same filters and parameter exploration for the current run within a particular timespan. She retrieves a cartouche labeled Big Machine run #0000042. She gestures the cartouche on top of a casier for visualization queries to extract the visualization filters and a playback of the parameter exploration. She then binds that information to a new cartouche for run #0000099.

The others at site Alpha are concerned because the phenomena they are observing is not progressing as predicted by relevant theories or as indicated by earlier experiments. Dr. Leslie quickly shows her colleagues the results from the previous run and is granted control of that sites visualization stream. She applies the filters she extracted earlier and rewinds the data stream a few hundred timesteps to gives Viz Corps Alpha a better picture of some anomalies within the simulation output. Dr. Leslie and her graduate student, Miguel identified some parameter tweaks in the scientific codes that may address the non-physical effects. Other 99

viz corps sites have taken notice of the Alphas activities and grant them control of the entire visualization stream. Miguel and Dr. Leslies filters and parameter exploration are applied to the entire Big Machine output stream, which convinces the Big Machine collaboratory to restart the Big Machine run with site Alpha input parameters. This move saves 100s of millions of CPU hours and leads to a number of important results.

B.3 Discussion

Is this scenario possible today? Its probably within the realm of possibility, but likely not feasible. The bandwidth requirements of such an application could be met with several optical network testbeds, provided there are transport protocols that maximally utilize available bandwidth. This typically is not the case with TCP whose congestion management algorithms interpret long round-trip times (RTTs) as congestion and throttle bandwidth. Several researchers, including colleagues at the CCT, are working on techniques to improve IO throughput to better utilize fast networks and high-performance storage systems. And of course the user interaction capabilities could be cobbled together using existing techniques for user interaction integration. But we cannot simply do more of what is done today and expect our systems to scale.

The scenario highlights three system aspects that pose obstacles to large-scale interactive ubiquitous computing applications. These are large numbers of independent nodes, resource heterogeneity and variable structures of data-flows between components within the large-scale system. Systems composed of large numbers of nodes imply both distributed computing components communicating over networks of varying geographical reach (from personal area networks to global scale networks). In terms of protocols and services for transporting interactive data, current day systems only support inter- active responsiveness for small number of nodes. In many cases, this is because interactive communication infrastructure is based on a centralized architecture. In the literature it is often hand-waved that these systems can be re-factored into decentralized architectures; but why have few interactive systems architects ventured in distributed interaction infrastructure waters? Even in the face of scalable remote distributed interaction communication protocols, synchronization and maintaining consistency across many interactive system components are still considerable challenges.

The research I present here helps enable such a future because it explores architectural approaches that decouple the definition of high-level blah-blah-blah behavior and structure from the specifics blah-blah-blah inter-component communication, a type of interactive system late binding. In other words, developers

would be able to build ultra-scale interaction applications that may use simple infrastructure for small-scale configurations and for large-scale configurations employ remote distributed infrastructure. Additionally, interactive system architects and engineers could more easily evaluate remote distributed infrastructure by conducting experiments in which they deploy and test the same application across different system configurations. Resource heterogeneity poses a problem to large-scale interactive system integration in that without generic APIs and services, such applications would require a priori knowledge of the specific resources to be employed within a large-scale interactive system. This requirement limits flexibility and increases complexity. Systems that provide abstractions that expose common functionality across inter- active resource instances while exploiting or mitigating pragmatic variations would address the challenges posed by interactive resource heterogeneity. Scenarios like the above-mentioned will continue to be infeasible if the considerable resources within a large-scale interactive system cannot be flexibly wielded and reconfigured to meet the varying needs of its human operators. Despite increasing I/O throughputs and computational capacity, realities of network distributed computing such as latency and unreliability will still exist. These system obstacles can be mitigated by strategies that allocate distributed interactive system resources based on on-the-ground conditions.

Vita

Cornelius Toole, Jr. was born in Mound Bayou, Mississippi, in 1981 to Cornelius Toole, Sr. and Alma Coleman Toole. After graduating as class salutatorian in 1999, he began studies at Jackson State University earning a bachelor's degree in computer science in 2003. Cornelius began graduate studies at Jackson State University and conducted research in distributed computing and scientific visualization under the advisement of Loretta Moore, Ph.D. and John Shalf (from Lawrence Berkeley National Laboratory). In 2005, Cornelius completed a thesis entitled, "Broker Services for Distributed Visualization." and received a master's degree in computer science. In 2005, Cornelius began doctoral studies at Louisiana State University in the Department of Computer Science. During this time, he was a graduate research assistant in the Tangible Visualization Laboratory and Scientific Visualization Group at the Center for Computation. He is currently a doctoral candidate in computer science and is completing research on software architectures for tangible user interface-applications in computing environments with distributed, heterogeneous resources.