Harmony Oriented Architecture

2011

Kyle A. Martin

University of Central Florida

Find similar works at: https://stars.library.ucf.edu/etd

University of Central Florida Libraries http://library.ucf.edu

Part of the Computer Sciences Commons, and the Engineering Commons

STARS Citation


This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
HARMONY ORIENTED ARCHITECTURE

by

KYLE A. MARTIN
B.S. Florida State University, 2006

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2011
ABSTRACT

This thesis presents Harmony Oriented Architecture: a novel architectural paradigm that applies the principles of Harmony Oriented Programming to the architecture of scalable and evolvable distributed systems. It is motivated by research on Ultra Large Scale systems that has revealed inherent limitations in human ability to design large-scale software systems that can only be overcome through radical alternatives to traditional object-oriented software engineering practice that simplifies the construction of highly scalable and evolvable system.

HOP eschews encapsulation and information hiding, the core principles of object-oriented design, in favor of exposure and information sharing through a spatial abstraction. This helps to avoid the brittle interface dependencies that impede the evolution of object-oriented software. HOA extends these concepts to distributed systems resulting in an architecture in which application components are represented by objects in a spatial database and executed in strict isolation using an embedded application server. Application components store their state entirely in the database and interact solely by diffusing data into a space for proximate components to observe. This architecture provides a high degree of decoupling, isolation, and state exposure allowing highly scalable and evolvable applications to be built.

A proof-of-concept prototype of a non-distributed HOA middleware platform supporting JavaScript application components is implemented and evaluated. Results show remarkably good performance considering that little effort was made to optimize the implementation.
# TABLE OF CONTENTS

LIST OF FIGURES ........................................................................................................... vi

CHAPTER 1: INTRODUCTION ............................................................................................ 1

CHAPTER 2: RELATED WORK ......................................................................................... 6

  Systems Architecture .................................................................................................. 8
  Improving Concurrency ............................................................................................... 12
  Phenotrophic Computing ............................................................................................. 14
  Poietic Computing ........................................................................................................ 15
  Spatial Abstractions ..................................................................................................... 17
  Harmony Oriented Programming .................................................................................. 22

CHAPTER 3: HARMONY ORIENTED ARCHITECTURE ...................................................... 26

  Concepts ...................................................................................................................... 26
  Programmable Architecture ......................................................................................... 36
  Scalability .................................................................................................................... 38
  Evolvability ................................................................................................................ 40
  Distributed Systems .................................................................................................... 41
  Development Lifecycle ................................................................................................ 44
  Case Study: Web-Based Gaming Startup ..................................................................... 46

CHAPTER 4: IMPLEMENTATION ...................................................................................... 53

  JavaScript Snippets ..................................................................................................... 54
  Asynchronous Execution ............................................................................................. 55
LIST OF FIGURES

Figure 1: Snippets, space, and diffusion regions ................................................................. 3
Figure 2: Singleton, multi-instance, and multi-spatial snippets........................................... 31
Figure 3: Point and region representations and corresponding diffusion graphs...................... 34
Figure 4: Web-based gaming architecture ............................................................................ 48
Figure 5: Harmony.js platform components. ....................................................................... 53
Figure 6: Event dispatching example................................................................................... 56
Figure 7: Database design.................................................................................................... 59
Figure 8: Snippet chains..................................................................................................... 61
Figure 9: Comparing performance with varying snippet chain length................................. 67
Figure 10: Detailed performance with varying snippet chain length.................................... 68
Figure 11: Comparing performance with varying link cost.................................................. 69
Figure 12: Detailed performance with varying link cost....................................................... 70
CHAPTER 1: INTRODUCTION

The future of computing will be dominated by ever increasing system scale and complexity. With the exponential growth of the Internet over the last few decades we have already created an integrated system of a scale never before seen. Unfortunately it is not used as such. It is carved into isolated sub-systems exposing limited functionality. Through a fragile patchwork of protocols, business agreements, and continuous effort, it has managed to avoid shutting down entirely.

Recent research on the design of ultra large-scale systems has begun to expose the limits of human abilities to understand and manage their complexity [1–3]. These limitations have shaped the development of software and the Internet and are why it is only recently a set of protocols and architectural paradigms were canonized to lay the firmament for cloud computing. Nevertheless, it too will fall far short of expectations due to human inability to manage the complexity of such large-scale systems. The only way to overcome these limitations is to remove humans from the equation entirely and instead rely on the system to design itself. Implementing a highly scalable platform for autonomous and evolvable intelligent systems capable of designing themselves is the ultimate means of coping with the complexity of ultra large-scale systems.
The most challenging aspect of implementing this platform will be ensuring its scalability and evolvability. Without those features implementing an ultra-large scale system is all but impossible. The effectiveness of all other facilities the system will depend on its ability to scale in complexity and capacity while evolving rapidly to adapt to changing usage patterns and functional requirements. This thesis presents a novel architectural paradigm for building such highly scalable and evolvable systems called Harmony Oriented Architecture. The challenges surrounding the implementation of an HOA platform are explored and a proof of concept implementation for JavaScript applications is evaluated.

The core concepts in HOA are derived from recent work in ubiquitous computing, software design, and scalable architectures that have utilized spatial abstractions to improve scalability and evolvability. In ubiquitous computing spatial relationships among nodes in a distributed system, such as their geographic location or coordinates in an abstract space, have been used to coordinate distributed computations in applications such as sensor networks and mobile ad-hoc networks [4–6]. Scalable architectures such as Space-Based Architecture [7], [8] and design paradigms like Harmony Oriented Programming [9], [10] utilize spatial data structures as a means of sharing state between decoupled application components and improving system scalability. HOP also introduces a number of concepts that are well suited to evolvable system architectures, such as strict isolation, state exposure, and representing coupling between components using their proximity within a virtual space. Harmony Oriented Architecture formulates these spatial computing techniques into an architectural paradigm for developing a platform for large-scale evolvable systems.
Figure 1: Snippets, space, and diffusion regions

HOA enforces strict isolation between components, called snippets, and prohibits all access to resources in other snippets, relying instead on a distributed spatial database to mediate interaction between snippets through a process of data diffusion. Each snippet is assigned a diffusion region in the spatial database (see Figure 1) and can only modify data stored within its diffusion region. Snippet state is kept entirely in the spatial database and exposed to any snippets within diffusion range. All modifications a snippet makes to data stored in the space are immediately diffused to neighboring snippets within diffusion range.

The strict isolation and state exposure in HOA significantly reduces coupling between application components and improves the scalability of the system. Representing functional relationships between snippets through their spatial arrangement improves evolvability by enabling message passing between components without the complexity and fragility of explicit interface dependencies. Snippets are highly decouple units of reusable functionality which can be assembled in particular arrangements to implement a desired behavior, much like how command line applications can be piped together in a variety of ways. This allows the overall functionality
of a system to be easily changed by simply altering the spatial arrangement of snippets in a space. A number of tools for developing and managing a large-scale system can also be readily implemented within an HOA platform, such as tools for testing and debugging, application deployment, and monitoring.

Given a collection of snippets it is possible to generate arrangements exhibiting desired behaviors using a variety of evolutionary computing techniques, potentially enabling autonomous evolution of large-scale systems. State exposure in HOA can be used to implement commensalistic software [11], [12] which uses the spatial abstraction to arrange host and symbiont snippets in proximity to each other to allow hosts to manage the execution of symbionts and detect abnormal behavior. Phenotropic computing techniques [13] can also be applied to allow snippets to observe the behavior (diffusion) of other snippets and reason about what they can do and how to interact with them. Using these types of general-purpose snippets along-side domain specific snippets provides the foundation for a self-designing ultra large-scale system.

An early prototype implementation of a non-distributed HOA platform for JavaScript application has been developed as the first step in a long-term effort to develop a general purpose HOA platform. JavaScript has been chosen for its history of providing secure and isolated scripting in web browsers and relative ease of use in this domain compared to other languages. Google's V8 JavaScript interpreter is used along with the Node.js JavaScript application server to provide an asynchronous I/O framework and standard library for JavaScript snippets. The
combination allows feature rich and relatively high performance snippets to be developed. The Generalized Search Tree (GiST) was used to provide spatial and non-spatial indexing for a simple non-distributed database. Experimental evaluation of this prototype will utilize a simple demonstration application consisting of a web server and a “chain” of snippets which process and relay requests along the chain to gauge the performance characteristics under varying loads.
CHAPTER 2: RELATED WORK

Recent research has shown that there are intrinsic limitations in human ability to design ultra large-scale systems. For such extremely complex systems, no single human or group of humans can ever understand the system as a whole. The scale of such systems becomes limited by the group’s ability to coordinate their efforts on the small parts of the system they can understand to achieve large-scale results. As groups become larger and federated systems begin to emerge cooperation and collaboration becomes much more difficult, limiting the overall complexity of the system, and leading to diminishing returns from additional scaling. Overcoming these limitations to build ultra large-scale systems requires self-designing systems that can manage the complexity of implementing applications in such a highly federated environment [1–3], [14], [15].

Building a self-designing system is not a simple task. It will require integrating concepts from a number of disparate fields, like software engineering, intelligent systems, and evolutionary computing. These systems will need to be able to observe and modify themselves to adapt their behavior based on changing usage patterns or explicitly specified requirements. They must be capable of reasoning about patterns in their current and past behavior to formulate plans for adapting future behavior to changing requirements. Executing these plans will involve both deterministic changes to the system for simple adaptations and non-deterministic evolution to
explore the space of possible changes and resulting behavior to discover complex adaptations.

While implementing these features will be difficult, the greatest challenge is designing a platform that can support them all at once. At the very minimum this platform must be highly scalable and evolvable to be suitable for use in an ultra large-scale system. It must also utilize abstractions that can effectively represent properties of the system to both humans and machines alike. All other facilities will leverage these traits to allow the system to grow and adapt to changing requirements. They are crucial to ensuring that the platform can be distributed across a multitude of machines to support massively concurrent applications while coping with the continual change and frequent failures that result from large-scale federated development.

A number of strategies for improving the scalability and evolvability of systems have been developed. Architectural techniques like Database-centric Architecture and Service Oriented Architecture and design techniques like event-driven development and Harmony-Oriented Programming seek to reduce coupling between application components to improve evolvability and allow components to be easily distributed across a number of machines and recovered after failures. Phenotropic and poietic computing techniques have been proposed to further reduce coupling and improve scalability through the use of intelligent system components that can observe and reason about other components to determine how best to interact with them and manage their execution. Spatial abstractions have also been used in a number of contexts to provide a simple and evolvable representation of system properties that avoid the complex graph-like representations of object-oriented techniques.
Many architectural approaches have been proposed to address scalability and evolvability issues in large-scale systems. The predominant techniques use distributed computing to achieve horizontal scalability and rely on reducing coupling between sub-systems to simplify distributed execution. There are several patterns shared by these techniques: databases are used to decouple system components from persistent state, and interfaces are used to decouple components from each other.

Database-driven architecture is by far the most popular architectural pattern and utilizes a central database that other system components access to persist and share domain state. Decoupling system components from domain state in this way simplifies the management of data in the system and allows components to evolve independently yet interoperate through the shared database. System components can scale independently from the database and the reduced coupling between components and persistent state helps improve the overall evolvability of the system. However, these databases quickly become the primary bottlenecks in the system and limit the scalability of the system.

Large-scale web applications have pushed the limits of database-driven architecture, which has led to a recent flurry of innovations to overcome the limitations of traditional database-centric architectures. Chief amongst these is the shift from complex relational databases to simpler and more scalable NoSQL databases. These databases are designed to take advantage of modern distributed computing techniques to scale horizontally and ensure availability. This
comes at the expense of certain guarantees provided by traditional databases, such as consistency or write performance [16].

Service Oriented Architecture has gained traction with the rise of large-scale web applications as it allows web applications to interact based on explicitly defined interfaces and policies which has facilitated the rise of the cloud-computing ecosystem [17]. These loosely coupled web services interact over HTTP using REST, XMLRPC, or similar methods that are widely used to implement SOA interface specifications. Design-by-contract allows cloud service providers to agree on functionality, interfaces, and management policies to allow for integrated ecosystems of services to form.

However, the scale of these ecosystems is inherently limited by the difficulties in achieving consensus in large-scale federated organizations. This leads to competing interface specifications that evolve independently to meet the needs of particular organizations and requires integration facilities within each autonomous system to allow for translation between competing interfaces. These integration facilities emerge organically when autonomous systems need to interact and are co-evolved as each autonomous system adapts to another’s independently evolving interface. Embracing an organic development model, rather than trying to avoid it through formal specifications, is the only way to manage the complexity of this inevitable co-evolutionary process [18], [19]. Architectural approaches like Component Based Architecture and Space Based Architecture can enable a degree of organic evolution that SOA cannot achieve.
In Component Based Architecture [20], [21], system components implement interfaces but these interfaces are not defined as part of top-down specification like that used in SOA. Instead, an ecosystem of co-evolved components emerges in which each component is adapted to the interfaces of other components in the ecosystem. Some of these components may be reused across multiple organizations while other may be built to meet the needs of one particular organization. One of the earliest examples of component-based architecture is pipelining between command-line applications. Each program is developed independently but over time a variety of tool chains emerge consisting of co-evolved programs that can be pipelined together to perform complex operations. Large-scale federated systems can be built from components using similar pipelining techniques so long as facilities are available to manage the organic evolution of components in the ecosystem.

Space based architecture utilizes a spatial abstraction to allow application components to interact indirectly through a “tuple space” [7], [8], [22]. Rather than explicitly defining a service interface or a graph-like pipeline, components query the tuple space to retrieve data inserted by other components matching certain patterns. This form of associative memory allows for highly flexible interactions between components that reduces complexity and improves scalability. Pipelining and services can even be implemented using specialized queries and tuple patterns to specify the pipelines or service interfaces. While this approach does provide a more general form of interaction between components it is simply a variation of database-driven architecture and provides no means of managing the ecosystem of co-evolving components.
Database-driven architecture decouples application components from persistent state but this often leads to an “impedance mismatch” between the state of running components and the state that can be persisted in the database. This is often referred to as the “object relational impedance mismatch” problem [23], but it is not limited to object-oriented development or relational databases since decoupling components from their state will always require translating between two representations of domain state. A variety specialized databases have been used to minimize the cost of translating between representations of specific types of data, such as object databases, document databases, and key-value stores [16], [24–26]. Persistent programming languages go even further and embed database features directly into a programming language to tightly couple the two and avoid translation entirely [27], [28].

Embedding application components directly into a database is one option to minimize the costs of decoupling that has yet to be considered. Application components embedded into the database can be provided with a rich set of APIs for accessing persistent state and interacting with the database while still enabling state to be shared amongst applications and allowing applications to evolve independently of the domain state. This approach can improve scalability and evolvability like traditional database architectures while achieving the tight coupling between components and state offered by persistent programming. Furthermore, application components become a part of the persistent state of the system, allowing components to observe and manipulate their properties directly. This level of transparency can provide greater insights into changes in the component ecosystem which can help guide the evolution of individual
components and opens up the possibility for autonomously adapting components to these changes.

**Improving Concurrency**

Non-blocking asynchronous execution models have been shown to maximize concurrency while minimizing the cost of threading and multi-processing on each node. Even with scalable execution models a system’s persistent storage model largely determines its performance as it is subject to strict physical limitations imposed by the underlying storage hardware. In this area highly scalable persistent data structures have emerged which maximize concurrency through the use of non-blocking lock-free isolation models [16], [29–33]. Traditional concurrency techniques have been shown to limit scalability due to the overhead of synchronizing large numbers of threads and processes and the reduced maintainability of complex thread-safe code. However, this complexity and overhead is only necessary to maintain the illusion of synchronous execution in an otherwise asynchronous execution model provided by the operating system. Asynchronous concurrency models have been shown to significantly improve scalability in highly concurrent I/O-bound applications by avoiding the overhead of synchronizing large numbers of threads and processes [34].

Unfortunately, most platforms have limited support for asynchronous concurrency models. System calls are typically blocking and usually only network related operations provide asynchronous calls. These platforms can simulate asynchronous operations by performing the
blocking operations in background threads while dispatching the results as events processed from a single-threaded event loop. Asynchronous applications are also notoriously difficult to program in traditional programming languages due to the decoupling of method calls from their results and scoped state. However, the significant performance improvements easily justify the added code complexity.

Some NoSQL databases like CouchDB and Redis use persistent data-structures and multi-version concurrency control to reduce the need for complex locking mechanisms. Persistent, log-structured, and append-only data structures all maintain multiple versions by ensuring that previous versions of the data structure are immutable and changes can only be made by appending a new version of the data structure. One of the fundamental limitations of persistent data structures is the large storage overhead and performance impact incurred when reading or writing a particular version. However, this can be mitigated through periodic rollover (writing out the latest version to a new data file) or by storing pointers to previous versions of nodes to reduce required storage (as in Redis). Despite this limitation, these types of data structures have been used extensively in the financial services industry and in functional programming languages like Clojure and Erlang [16], [33].

The greatest advantage of using persistent data structures is the significant reduction in concurrency-related complexity gained through Multi-Version Concurrency Control which requires only a single write lock to serialize write operations and provides lock-free read operations. Operations only need to store a reference to the version of the data structure at the
time when they are initiated to ensure a consistent view. With such a simple concurrency model highly scalable applications can be built supporting a large number of concurrent accesses to the same data structures [29], [31–33].

**Phenotropic Computing**

The concept of phenotropic computing has been proposed as a solution to overcoming the limitations of traditional software development that inhibit evolvability[13]. Phenotropics “roughly translates to 'surfaces relating to each other'.” It is inspired by the “fuzzy” statistical techniques used in robot navigation, machine vision, and related fields in computer science which must design software that interfaces with the physical world. In these fields, the precisely defined structures typical of traditional software development fail miserably when confronted with the unpredictable streams of data coming from the sensors in the real world. The only way to accommodate such unpredictability is to dispel the usual notions of mathematical precision and aesthetics that lay at the foundation of computer science. The critical flaw with traditional approaches to software development is that even the slightest variation in a program's code can have disastrous effects, such as a single bit error in a bad memory module crashing an entire operating system.

Relying on “fuzziness instead of perfection”, phenotropics attempts to apply the same statistical techniques within the “internal world” of the computer system that have been successful at interfacing with the physical world. An example of this technique is described in
the case of the Embrace system that sought to allow GUI applications to utilize the user interfaces of other applications directly as though a virtual user was “looking” and “interacting” with them. Without the need for strict interface specifications, these applications could observe the interface provided by another, and just like a human, make assumptions based on experience as to how to interact with the application to achieve a desired behavior. A simple example describes an application that needs to perform an arithmetic computation by using another calculator application just as any other human user would. A similar but far more limited concept has emerged recently in the form of web mash-ups that typically exploit undocumented APIs and “screen scraping” techniques to extract data from and interact with web application in similar ways to humans.

**Poietic Computing**

In several related papers [11], [12], [35] autopoietic and allopoietic techniques for autonomous software management and evolution are described. Autopoiesis literally means “self creation” and in contrast, allopoiesis means “creation of something other than itself.” The process of creating software is allopoietic, and in most cases, the software itself is allopoietic and creates something other than itself, such as a document, user interface, etc. Autopoiesis on the other hand is a characteristic of cellular life and the reproductive process. Autopoietic software can manage and “create” itself to adapt to changing circumstances, like recovering from failures, or evolving new functionality.
The use of allopoietic and autopoietic software has been suggested as a means of overcoming human limitations that impede the development of ultra large-scale systems. These concepts can be used to allow software to take “[responsibility] for itself and its own future by participating in its own installation and customization, maintaining its own health, and adapting itself to new circumstances, new users, and new uses.” [35]. Such “conscientious software.” can be implemented using the “fuzzy” techniques described above, like machine learning and digital evolution. In an ultra large scale system these computationally intensive tasks can be performed using the “spare” processors that are becoming increasingly available with the proliferation of multi-core processors and the current lack of suitable software engineering techniques which can exploit the new found parallelism in the average desktop or server.

Conscientious software has been extended to incorporate the biological concept of commensalistic symbiosis in [12]. This concept is best described through the example of the sea anemone and the clown fish. Here the sea anemone acts as a host species which is capable of surviving on its own and defending itself from predators using poisonous tentacles while the clown fish is a symbiont which has evolved immunity to the anemone’s poison allowing it to live in the protective “forest” of the anemone’s tentacles. The sea anemone neither benefits nor is harmed by the presence of the clown fish. However, the clown fish has no defense against predators outside of the anemone's protection and does gain from the relationship.

The notion of a neutral host that provides an advantage to a symbiont is applied to the autopoietic and allopoietic software model proposed in [35]. The allopoietic software
components are the “defenseless” symbionts which can only survive (execute safely) within the protective confines of the autopoietic host which is highly adapted towards maintaining its own well being (recovering from crashes, detecting degraded states, etc.). Autopoietic hosts are also capable of communicating with each other through a “nervous system” of linkages to manage the execution of allopoietic symbionts, including the loading/unloading of symbionts, restarting failed symbionts, etc.

Spatial Abstractions

Spatial abstractions have been used to address scalability and evolvability issues in the areas of ubiquitous computing, software design and software architecture. In ubiquitous computing, the techniques attempt to simplify the development of distributed applications running on limited hardware platforms distributed throughout a physical space. Libraries, middleware, or specialized programming languages provide an abstraction of the underlying platform and spatial relationships amongst devices that improves scalability and simplifies the development of complex applications. Design approaches utilize virtual spaces to represent the relationships between software components. Changing the arrangement of components in the spaces allows application functionality to be adapted and evolved without modifying the components themselves. Architectural patterns for improving the scalability of distributed systems have used multi-dimensional spaces to share state amongst system components.


**Ubiquitous Computing**

Spatial abstractions have been used in several approaches in ubiquitous computing to hide the underlying network complexity. One such approach utilizes “Smart Messages” consisting of code and data segments that are distributed throughout the network from one node to another [5]. Each SM is capable of routing itself from one node to another by executing associated code on the node. Each node in the network provides a Java virtual machine and a distributed virtual machine (DVM) emerges through the propagation of SMs between nodes. This DVM provides abstract spatial references to allow SMs to propagate themselves towards regions of interest without needing to know which devices are within a region or how to route the SMs to the destination. However, accessing specific devices, especially mobile devices, is far more difficult and requires searching through regions and possibly the entire space. This technique can also utilize virtual spaces since coordinates for devices and spatial regions can be arbitrarily assigned, although routing within an ad-hoc network will be more complicated if the virtual space is vastly different from the geographic locations of the nodes.

Tuples Over The Air (TOTA) is a distributed computing framework that hides the network complexity behind a spatial abstraction [4], [36]. Using this abstraction tuples consisting of a set of content fields and propagation rules are routed through the network based on the spatial proximity of tuples and nodes in the abstraction. The propagation rules govern how the tuples are routed throughout the network including both the distance tuples should propagate from the source, how the distribution is affected by other tuples in flight, and how tuples are modified as they are propagated. In a sense, each tuple is like a mobile agent capable of
controlling its own movement through the network. What emerges is a “field” of in-flight tuples that are periodically updated to reflect the changes to the spatial structure as mobile nodes move throughout the space. Tuples are buffered at each node long enough to allow them to be re-propagated as new nodes join the network or the spatial topology changes. Complex routing mechanisms can also be used to implement virtual spaces using techniques such as content-based routing. From the perspective of a developer, TOTA reduces the network into simpler spatial data structure in which it is possible to navigate the space and observe nearby tuples as well as modifying the space by injecting new tuples.

The Proto language for programming distributed sensor networks transparently abstracts the underlying network behind a spatial representation described as an “amorphous medium” [6]. Data in this medium propagates over time like a field occupying every point within the space. Applications written in Proto are compiled into sequences of operations executed on all devices which take into account propagation delays to allow for synchronized behavior. The language also includes several constructs for restricting the range of an operation, setting delays for values to “establish state using feedback loops”, and several operations that retrieve, enumerate, and aggregate neighborhood values. One of the obvious weaknesses of Proto is that it is optimized specifically for networks of embedded devices with limited hardware resources, which limits the widespread adoption of this approach or scaling it to support applications that are more complex.
Space-Based Architecture

Space-Based Architecture uses tuple spaces to implement scalable distributed applications executed over a grid-like network of independent processing units [7]. Tuple spaces are distributed collections of arbitrary tuples which allow nodes to concurrently insert tuples and retrieve tuples matching various patterns. The Linda programming language introduced the concept of tuple spaces and a number of middleware platforms implementing tuple spaces and distributed processing and management facilities are available for a variety of other languages [8], [22].

Tuple spaces are somewhat analogous to spaces in HOP however they only serve as a shared data structure to facilitate state transfer between decoupled application components running on different processing units. In HOP spaces also mediate the interactions between components through diffusion and there is strict isolation between application components. In SBA traditional distributed processing models (master/slave, etc.) and messaging facilities are used to manage the execution of application components and allow components to interact directly with each other to coordinate access to the tuple space (passing tuples between components).

Software Design

Collaborative diffusion is an “anti-object” design paradigm in opposition to top-down decomposition traditionally used in object-oriented design [37]. Object-oriented decomposition
can lead to acceptable solutions but the process blinds the designer from other more effective
designs. Avoiding object decomposition entirely and rather using an anti-object technique is
shown in several case studies to produce a much more effective solution. One study applies the
technique to game AI design by utilizing the state of a background objects like the game space to
facilitate planning the paths of AI controlled foreground objects. By diffusing state through the
game space, path planning simplifies to a gradient descent problem. Collaborative diffusion
relies on the availability of an underlying spatial data structure in which application state can be
diffused and exposed to other objects.

The Silhouette visual language presented in [38] applies the “anti-object” concept to
software design. In Silhouette, an application is designed using a set of loosely defined nestable
objects representing the underlying functionality and structure of the application. Designers have
full control over the mapping from visual representations to high-level language abstractions
allowing for the composition of software by manipulating its spatial representation. The resulting
shape conveys more information about the complexity and connectivity of application
components than a traditional visual design language like UML. Such a malleable design is also
highly evolvable as small changes to the design can significantly alter the underlying language
abstractions.
Harmony Oriented Programming

Harmony Oriented Programming is a concrete “anti-object” programming paradigm that is heavily influenced by eastern philosophy [9], [10]. Western philosophy views the world as a “static and unchanging collection of objects that can be described and analyzed through categorization and formal logic.” In contrast, Eastern philosophy “does not focus on objects and their attributes, but rather considers the broad context and defines the world in terms of harmony, context, and resonance ... a mass of continuously interacting substances rather than a collection of discrete objects.” Just as object-oriented design has its roots in Western philosophy the authors have devised a design paradigm based on Eastern philosophy that resembles in many ways the spatial computing approaches described earlier.

The authors describe five principles based on the Eastern concepts of harmony, context, and resonance:

- “Code Exposure” avoids the rigid encapsulation of the object-oriented paradigm by decomposing programs into pieces called snippets without providing well-defined boundaries or interfaces for the code in those snippets. Snippets are arbitrary pieces of code written in SmallTalk.
- “Code Positioning” assigns one or more positions within a virtual space to all snippets and spatial proximity allows related snippets to share data through diffusion.
- “Information Diffusion” disperses data generated by any snippet of the program within a limited region around that snippet.
• “Information Sharing” all state is owned by the space and equally accessible to any proximate snippets, i.e. no encapsulation.

• “Balance” describes the property that “all data produced by one part of the program is consumed by one or more other parts of the program.”

Together these five principles provide a concrete anti-object paradigm. These principles also establish a notion of “fields” of diffused data similar to the fields described in the TOTA and amorphous computing approaches. A snippet and the fields it emanates form a “substance” which provides a similar notion to an object but without the rigid and fragile encapsulation required by the object-oriented paradigm. The ultimate motivation for Harmony-oriented programming is to enable more fluid software evolution by allowing these malleable “substances” defined in a space to be repositioned and molded by programmers to effect changes in the software. By avoiding the frailties and limitations of the object-oriented model the authors have produced a programming paradigm in which major changes to the behavior of a program can be achieved with limited modification to the “real” code contained in snippets.

Beyond the similarities to spatial computing techniques previously described this paradigm bears a striking resemblance to a spatial database. Snippets execute in isolation from other snippets and communication between snippets is only possible by diffusing data through the virtual space. This is very similar to different database clients inserting data into a spatial database to communicate. To achieve “balance” only snippets that lie within the diffusion region will have access to the diffused data, similar to a continuous spatial query being triggered when
matching data is inserted. Existing spatial database techniques and implementations could be leveraged to support a similar system which could provide a much more useful platform for facilitating software evolution. The high degree of decoupling between snippets means they can potentially be executed on any number of nodes in a distributed system provided a means for distributed diffusion is available, such as a distributed spatial database. Such a platform may be more a general-purpose solution for spatial computing than the previously described approaches and could be adapted to support a variety of applications.

HOP appears to have been developed as a more general-purpose platform for implementing commensalistic applications. Host-symbiont relationships can be easily implemented through a combination of snippets with fields arranged in such a way that a symbiont snippet can be “encapsulated” and managed by one or more hosts that interact with each other directly through diffusion. By implementing a privileged snippet management interface made available to host snippets it should be possible to have multiple interrelated symbionts (in close proximity) managed by one or more host snippets which diffuse state to and from the symbionts to manage their execution. By incorporating such a technique in a distributed harmony-oriented platform, large-scale commensalistic systems should be feasible.

The concept of phenotropic interactions between application components through “fuzzy” interfaces is reflected in the Harmony-oriented paradigm. The chief limitation precluding widespread use of phenotropic techniques has been the computational complexity and overhead associated with machine learning and similar fuzzy techniques required for a
component to be able to reason about the interfaces of other components. Harmony orientation provides a relatively simple spatial abstraction that is easily comprehensible to both humans and machines, which can reduce the cost of these fuzzy techniques. However, until these techniques become more prevalent, human directed construction of component interfaces will still be necessary. The Harmony-oriented approach is a step in the right direction in that it allows those manually designed interfaces to be fuzzier than object-oriented counterparts through the “fields” emanated by snippets and the malleability of the “substances” in the virtual space. Snippets with pattern recognition and learning capabilities could be implemented to observe other snippets in the space through their corresponding fields and determine what those snippets are capable of how to interact with them.
CHAPTER 3: HARMONY ORIENTED ARCHITECTURE

Concepts

The central concept in Harmony Oriented Architecture is the same as in Harmony Oriented Programming: the use of a spatial abstraction to mediate the interactions between application components and enforce isolation and decoupling [9], [10], [39]. Spaces contain spatial objects representing application components called snippets. Each is decoupled from other snippets in the space and executed in complete isolation. Snippets can only interact indirectly by diffusing data through the space to proximate snippets. The spatial arrangement of snippets implicitly specifies the coupling between them rather than relying on explicit references to component interfaces. HOP presents this concept as a programming language extension and design paradigm. In HOA, this concept is explored as an architectural paradigm for large-scale systems.

HOA is not concerned with how individual applications are written, but how many different application components can be composed into large-scale systems using the spatial abstraction presented in HOP. This is achieved by tightly integrating application server functionality into a distributed spatial database server. The applications are directly managed by the database and executed in complete isolation like snippets. They could be seen as complex programmable continuous queries as they are asynchronously executed to handle database events.
(like diffusion) and can modify the database to trigger additional events that may affect other CQs (snippets). Here we briefly describe the original concepts presented in HOP, their extensions in HOA, and other related concepts.

**Spaces**

In HOP, a space is essentially a global spatial data structure with access controls that restrict snippets to viewing only proximate data. Since a snippet is just an isolated segment of Smalltalk code with no associated data when it is executed the proximate data in the space is brought into the scope of the snippet's code. Any data created by the snippet is also stored in the space and becomes available to any other snippets within diffusion range. A simple 2D grid was used with each cell storing references to diffused data. This design clearly cannot scale to support large spaces with large numbers of snippets. More efficient spatial data structures are needed to quickly look up the data objects within proximity of a snippet. This would essentially merge spatial database facilities directly into the language.

In HOA, a space is effectively a distributed spatial object database storing serialized objects in a format that can be deserialized by snippets implemented in a variety of languages. Each snippet is allowed to store and retrieve objects within its diffusion area. An API is exposed to each snippet that allows it to query the database directly for spatial objects within diffusion range.
To avoid the overhead of a large number of snippets periodically polling the database a distributed event-dispatching mechanism is needed to push events to affected snippets. Spatial event handlers can be registered in snippets to receive notifications for events occurring within diffusion range, such as newly diffused data or changes to a neighboring snippet's shape or location. When such changes occur the spatial database is updated and a spatial event is dispatched to snippets within the diffusion range of the originating snippet. The event dispatcher performs queries on the spatial data structures to determine snippets in range to receive the data and enqueues events to be dispatched asynchronously, possibly to a snippet running on a remote system. When the local event queue is processed, each event is dispatched to affected snippets by invoking registered callbacks in each snippet to handle the event. The event dispatching facility can be used for a variety of spatial events like diffusion or changes to properties of spatial objects (location, shape, etc.). General-purpose non-spatial events like system-wide events (restart, shutdown, etc.) could also be easily supported.

Scalable design techniques such as asynchronous operations, persistent data structures, and P2P distributed computing will be necessary to support spaces containing large amounts of data and numerous snippets distributed across a network. A number of factors will affect the performance of the system, such as data and snippet placement, overlay network topology, distributed indexing methods, etc. Future research will evaluate the various strategies and their performance in this domain.
Snippets

HOP implements snippets as arbitrary segments of Smalltalk code that do not store any data locally and cannot reference any code defined in other snippets, only libraries and modules available in Smalltalk and accessible to all snippets. All state is stored in the space within the diffusion region of a snippet and is mapped into the scope of the snippet's code when it is executed. Since all of a snippet's state is stored in the space, it is available to any neighboring snippets within diffusion range to access or modify. Snippets could be seen as a special form of object that has strictly encapsulated functionality and strictly non-encapsulated state, minimizing coupling but still allowing state sharing between objects. Potentially HOP could be implemented in any language however few languages natively support the degree of isolated execution and sandboxing needed to securely execute snippets. This relegates HOP to niche languages or extensions of other languages, such as the Harmony Oriented Smalltalk implementation described in [39].

A snippet in HOA can be implemented in potentially any language by using multiple processes or language level facilities to enforce isolation. Resources are provided by the HOA platform through a standardized Snippet API that exposes the database and event dispatching facilities of the platform to allow the snippet to interact with proximate snippets in the space through diffusion. Snippets will also have access to the system resources and standard libraries available through the language they are implemented in unless they are executed in a sandbox for security reasons. The Snippet API is the only way for snippets to interact with each other, though it is still possible for snippets to interact through facilities provided by their language, such as a
shared file. This high degree of isolation and decoupling also enables interaction between snippets implemented in different languages with their own implementations of the Snippet API.

**Snippet Instances**

Relationships amongst components of large-scale complex systems may not be easily projected into a space of n-dimensions but this may be beneficial as it prevents the design of overly complex systems. However in cases where non-local coupling is absolutely necessary it can be achieved through *instances* of a snippet distributed throughout the space, all effectively pointing to a single running snippet. Instances are exposed through the Snippet API and can be referenced in spatial operations to limit them to a specific instance's diffusion region. Singleton snippets can be simply implemented as a snippet with a single instance which is used by default in spatial operations.

Snippet instances can also improve performance in cases where a number of snippets are executing similar code and the application domain does not require strict isolation. The snippets can then be replaced with instances of a single snippet whose code has been modified to handle any instance specific requirements. Such optimizations are especially necessary when the snippet language requires additional overhead to enforce isolation for each snippet, such as execution in separate process or storing interpreter state.
Figure 2: Singleton, multi-instance, and multi-spatial snippets.

Snippet instances can also be placed in different spaces to allow for inter-spatial coupling, allowing spaces to be used as large-scale components of the system coupled through multi-spatial snippets. However, such coupling will significantly complicate distributed execution since multi-spatial snippets will simultaneously access multiple spatial databases. Limited use of multi-spatial snippets as “bridges” between different components should simplify placement but there can be no guarantee that all needed spatial databases will be locally available. In Figure 2, snippets, instances, and multi-spatial snippets are depicted.

Dynamic Snippets

In some applications, it may be useful to have snippets that frequently change their spatial properties in response to rapidly changing functional requirements or as a representation of domain state. For instance, a snippet may place multiple instances of itself within another space to represent the geospatial coordinates of moving objects in a physical space. Each instance can then send and receive events to proximate snippet instances in the space or observe
geospatial data stored in the space. Effectively these instances are continuous queries over regions of the space and their tight integration with the database can allow for near real-time event notification and query modification.

Snippets that can frequently change their location or shape within a space will require specialized indexing and event dispatching mechanisms to reduce the overhead of updating snippet spatial properties. This can be achieved in a number of ways, such as using dedicated spaces optimized for dynamic snippets, or using specialized indexes over subsets of the snippets in a space. Supporting dynamic snippets will also significantly complicate placement algorithms. However techniques already used in applications with frequently updated spatial objects, like location based services and massively multi-player games, could be used to optimize the placement of dynamic snippets.

**Multi-Lingual Integration**

Conceivably any language could be used to implement a snippet. A snippet API would be exposed in each language to facilitate interaction between snippets implemented in different languages. However, such a general approach is far from practical with current platforms and even less so when support for executing untrustworthy snippet code is desired. If untrustworthy code is not an issue then separate processes for each snippet's interpreter or compiled code can be used to enforce isolation. IPC can then be used to expose the snippet API to the snippet running in each process. Ideally, operating system level support for intra-process isolation would be available to enable sandboxed execution of compiled code. Individual threads could then be
executed in mutually exclusive segments of the process address space that interact with each other and shared resources using asynchronous mechanisms provided by the operating system.

Sandboxed execution of compiled code has been shown to be possible using Google Chrome's Native Client library [40] but is limited to specific platforms and has considerable overhead. Lacking other options for sandboxed execution of compiled code within the same process, an interpreted language that does provide such an execution model is necessary. JavaScript is one such language and as the de facto standard scripting language of the Web has a proven history for providing secure sandboxed execution of untrustworthy code.

**Diffusion**

Diffusion is similar in many ways to message passing between objects in languages like Smalltalk. However, snippets cannot send messages to other snippets directly and can only interact indirectly by diffusing data into the space containing them. This data then “flows” to other snippets in proximity based on the current arrangements of snippets in the space. This indirect form of message passing enforces isolation and decoupling and allows for such things as commensalistic host snippets which can manage symbiont snippets within range of its instances or side by side testing of different versions of a snippet to automatically detect regressions by comparing their diffusion patterns.
For any given configuration of snippets in a space a diffusion graph can be generated that represents the flow of diffused data between snippets. These graphs may either be directed or undirected depending upon whether snippet instances are represented in a space by a point or region. Undirected diffusion is likely to be most common but directed diffusion may be useful for cases where a snippet must be able to observe other snippets without being observed by them, such as a commensalistic host snippet observing one or more symbiont snippets.

![Points and Regions Diagram](image)

**Figure 3**: Point and region representations and corresponding diffusion graphs.

Using points to represent the location of snippet instances requires the diffusion region to be generated for each proximity query based on a radius or other properties specified by the snippet instance. Only those snippet instances whose locations lie within the generated diffusion region will be retrieved by the proximity query. One-way diffusion between snippets is possible in cases where a snippet lies in another’s diffusion region but doesn’t contain the other snippet within its diffusion region. This leads to directed diffusion graphs as depicted on the left diagram in Figure 3.
If diffusion regions were used to represent snippet instances in a space then proximity queries would instead need to search for overlapping regions rather than contained points. This leads to undirected connectivity graphs as shown in the right diagram in Figure 3 (notice how the purple and green snippets are connected via overlap but not via containment). Using both representations within a space is possible if proximity queries search for both point containment and region overlap simultaneously.

With either representation, changes to the spatial configuration of the snippets can significantly alter the flow of diffused data and the resulting behavior of the larger system. For instance, if the blue and purple snippets in Figure 3 were moved slightly further away from the green snippet the bidirectional flow would become unidirectional using a point representation, even further and the connection is lost completely using either representation.

Diffusion graph may appear similar to the graph-like relationships in object-oriented programming but they are very different. Foremost amongst these differences is the continuous nature of the spatial arrangements that allows for significant variation without changing the resulting diffusion graph. Spaces can have arbitrarily high dimensionality and become effectively graph-like but such spaces are hard to comprehend, for humans at least, and simpler lower-dimension spaces will likely be more common. Diffusion graphs in low-dimensional spaces also help enforce decoupling by making it harder to represent non-local interactions between snippets.
There are three forms of diffusion available to snippets. Persistent diffusion inserts data into the spatial database and dispatches a diffusion event to proximate snippets. Event diffusion only requires the space to store the data for as long as it takes to dispatch it to proximate snippets. Ephemeral diffusion is like event diffusion but also specified a short lifetime for the diffused data to be persisted and available to snippets that move into proximity. This is useful in dynamic spaces as it allows moving snippets to see “trails” of data that has recently been diffused in a location.

To facilitate data interchange between snippets in different languages diffused data should be of the simplest possible format that can be easily shared between different languages. Standardized serialization formats like XML or JSON are preferred for their relative simplicity and widespread availability of existing parsers.

Programmable Architecture

Harmony Oriented Architecture is most similar in concept to Space Based Architecture. Both use a spatial abstraction to mediate interactions between components to improve evolvability and scalability but HOA also represents the relationships between components using the spatial abstraction. In SBA the relationships are implicitly defined through the types of tuples inserted by components and the queries used to retrieve tuples from the space. Explicitly representing these relationships within the space allows components to observe and manipulate
those relationships directly. This also enables a form of “programmable” architecture not available in other architectural techniques.

The generality and simplicity of the spatial abstraction used in HOA allows for other architectural approaches to be implemented using a HOA platform. Snippets are simpler than either components or services as they are stateless and do not provide any interfaces accessible from other snippets but they still encapsulate functionality and enforce decoupling. Snippet interfaces can be simulated through patterns in diffused data, much in the same way that tuple patterns can define the interfaces between components in SBA. Just like components, snippets evolve independently and must adapt these patterns through co-evolution. These patterns may also be formally specified using design-by-contract to implement services using snippets. These specifications may also be stored in the spatial abstraction in a format that allows snippets to observe and modify the patterns, which could enable self-designing services.

Although interface driven development techniques can be used in HOA they will increase coupling between snippets, which will limit scalability and evolvability. However, allowing snippets to specify the patterns of data they will send and expect to receive can be useful for documenting snippet behaviors and facilitating phenotropic interactions. This can also help to optimize the event dispatching process by allowing diffused data to be filtered by the patterns to avoid the cost of unnecessary diffusion. However, reasoning about diffusion patterns will be more difficult than reasoning about the spatial relationships between snippets in a space, which will reduce comprehensibility and increase the complexity of phenotropic snippets.
Unlike other architectural approaches, HOA provides an architectural “programming” model that allows component relationships to be specified using a simple spatial abstraction that components themselves can observe and modify to facilitate self-design. A visual design process similar to that used in Silhouette [38] can be applied at the architectural level to specify the relationships amongst system components and other high-level features, such as the allocation of components amongst cloud-hosting providers. These visualizations could also be used to store documentation, like system requirements, developer assignments, implementation notes, etc. Reusable architectural patterns can be captured as a set of spatial relationships and diffusion patterns amongst snippets and applied in various ways to facilitate rapid architectural development.

**Scalability**

One of the primary limitations of the HOA approach is the potential performance impact from executing code within the context of the database. The naive execution method would be to assign threads to each snippet and execute them concurrently alongside database specific threads but this method does not scale well due to the overhead of system threads and locking. Most embedded interpreters currently available do not adequately support multi-threaded execution without significant performance penalty and executing compiled code in isolation requires multi-processing even with an intra-process sandboxing framework like Native Client [40].
Using “green” threads or an asynchronous, non-blocking, event driven architecture is the next best approach to executing both interpreted and compiled code alongside one another. This technique allows a single process to handle the execution of code in multiple interpreters running in dedicated threads or processes. Any blocking operations will have to be handled by background threads to avoid blocking within an interpreter. Compiled snippets are easily executed within this model using their own dedicated threads or processes. The greatest drawback with this approach is that malicious or defective snippets, whether compiled or interpreted, can potentially block the thread running the event loop and stall the entire system.

A multi-process method is the best approach to executing both interpreted and compiled snippets within an event driven model while preventing malicious or blocking snippets from disrupting the event loop. In this model, the main process will manage the execution of database operations while child processes will manage the execution of snippets. Each child process has its own event loop that manages the execution of one or more interpreted or compiled snippets. Multiple threads may be used within each process to handle local blocking calls and to minimize the effect of blocking or malicious code. However, most interpreters are single-threaded which can allow a malicious snippet to block the event loop. Remedying this would require terminating the blocked process and restarting affected snippets in a new process. Snippets are stateless and decoupled so restarting them bears little risk of corruption, provided any private snippet state had been successfully persisted before they are restarted.
There are three forms of evolution that can occur in an ultra large-scale self-designing system: engineered, anthropogenic, and autonomous evolution. Engineered evolution is any change in the behavior of a system that was planned and executed by humans, i.e., not the result of a self-design process. Anthropogenic evolution occurs when the system adapts to humans based on changing usage patterns and explicitly specified requirements, while autonomous evolution occurs independent of human activity to proactively “discover” useful adaptations. Harmony Oriented Architecture is designed to provide a platform that can enable all three types of evolution to take place.

HOA is based on principles that were originally devised to reduce the costs associated with engineered evolution. The high degree of decoupling and isolation enforced by HOA enables application components and their relationships to be easily modified while minimizing the risks associated with failures. Utilizing a spatial abstraction not only improves decoupling but also improves intelligibility due their reduced complexity and innate human ability to process spatial relationships. HOA has the flexibility to support traditional engineering approaches without much additional overhead and the reduced complexity of the underlying platform greatly improves manageability. Together these properties reduce the cost of engineered evolution and are crucial to supporting anthropogenic and autonomous evolution in ULS systems.

Widely used techniques in autonomous computing, machine learning, and evolutionary computing can be implemented in HOA to enable anthropogenic and autonomous evolution [41–
These methods can be used to automatically adapt the behavior of a system to such things as changes in load, usage patterns, or system failures. Requirements specifications can also be supplied to direct evolution towards specific goals. Using autonomous evolutionary processes the system can enable “experimentation” to discover optimizations or novel behavior independent of human activity or guidance. Different portions of an ULS system will evolve independently and no matter whether the evolution of each part is engineered, anthropogenic, or autonomous, co-evolution will occur as each part evolves to adapt to the changing behaviors of the rest of the system. This form of emergent co-evolution will allow a ULS system to design itself from the bottom-up and avoid the complexity of implementing a holistic top-down design process at such a large-scale.

**Distributed Systems**

The high degree of decoupling between snippets and their state-less nature is very useful in a distributed architecture as it allows for flexible assignments to hosts for execution, mobility between nodes, parallel execution, and fast failure recovery. Data placement strategies used in distributed systems to replicate and cluster data to minimize network overhead can be leveraged to replicate and cluster snippets and data based on their spatial proximity. Snippet replication also enables parallel execution of a snippet by allowing each replica to concurrently access the state of the snippet stored in the distributed database.
Snippet replication can be seen as a special case of snippet instantiation in which each snippet instance is associated with multiple replicas that concurrently access the instance state. Just as an API must be provided to allow a snippet to manage its instances, a similar API must be provided to allow a snippet to manage its replicas and coordinate their execution using private event channels. Diffusion through instances of replicated snippets can involve either dispatching the same event to all replicas or dispatching an event to only one replica at a time using a load balancing technique. Which option to use can be specified in the instance configuration retrieved when the spatial database is queried for instances to receive the diffusion event.

Distributed event dispatching is required to facilitate diffusion between snippets running on different hosts. Like the spatial computing techniques for distributed systems described in [4–6], distributed event dispatching hides the underlying complexity of the network from snippets. This both significantly simplifies development and improves scalability since snippets executed on the same machine or on machines spread across the Internet use the exact same API to diffuse events.

By abstracting the underlying network HOA can significantly reduce the complexity of implementing distributed computing algorithms. For example, a snippet implementing a MapReduce operation can be instantiated in proximity to other snippets so that they may simply diffuse an event to initiate a MapReduce operation. Based on parameters passed with the event the MapReduce snippet instantiates itself in proximity to required partitions of the data set and replicates itself to each node holding one or more of these partitions. Replicas then perform the
map operation on local partitions, return results through the private event channel to replicas chosen to perform the reduce operation, and then finally the results of the reduce operation are diffused through the main instance of the MapReduce snippet which received the start event. The flexibility provided by HOA also allows a number of other implementation options, such as using distinct snippets implementing the map and reduce operations that are coordinated using a controller snippet. Other distributed computing algorithms that do not require explicit knowledge of the underlying network can be implemented in similar ways.

For those algorithms that exploit the underlying network topology, there are a variety of techniques for exposing the topology to snippets. In the simplest case, snippets can determine the topology by communicating directly with each other using networking facilities provided by the snippet language. The most complicated option is using a specialized graph database managed by the system to store a detailed representation of the topology and the distribution of spatial objects across the network. Simpler representations of the topology can be implemented using spatial objects to represent nodes in the network. These objects could be clustered based on network distance or explicitly placed in spatial partitions representing relationships amongst nodes. Snippets implementing a distributed computing algorithm can then instantiate themselves in the space to interact with the topology abstraction.

Using a spatial abstraction also has the benefit of allowing snippets to easily manipulate the topology of the network and their placement on nodes. Snippets can be instantiated in proximity to nodes to represent their current placement and if a snippet wishes to move itself to
another node, it can simply place its instance into proximity of another node to initiate a migration. Explicitly partitioned spaces can be used to represent subsets of the network with particular properties, like nodes assigned to a particular cloud-hosting provider, or nodes to be used for particular tasks. Snippets can then move nodes between partitions to alter these properties, such as migrating a node between cloud-hosts or increasing the number of nodes assigned to a load-balancing group.

**Development Lifecycle**

A large-scale application can be managed from inception to retirement using tools implemented as part of a HOA platform. Version control, issue tracking, documentation and code annotation, and other development tools can be easily implemented and integrated into the larger production environment to facilitate continuous development.

**Code Database**

One advantage of HOA is that snippet code, whether implemented in an interpreted or compiled language, can be stored in the database to ease development and enable distributed deployment. The simplest case is with interpreted code since the text can be stored directly in the database, possibly in a compressed form. For byte-compiled and JIT compiled languages the binary versions can be cached in the database alongside the original text with additional metadata to support platform dependent binary formats. For compiled languages there are two options, either the source code can be stored in the database to allow for background compilation as
needed, or pre-compiled binaries can be stored with the requisite metadata to identify target platforms. The latter option requires a similar build process to traditional deployment strategies whereas the former is similar to the case with byte-code and JIT compiled languages.

Packaging

HOA applications can be packaged for deployment on individual nodes in a number of ways. A HOA application can be compiled into a single executable using a HOA runtime library provided all snippets could be compiled to a binary format. With purely interpreted snippets, it is more difficult to assemble a single executable but it is still possible through embedded interpreters and storing compressed code in the executable. Single executable deployments are better suited for traditional server applications and user-facing applications on a desktop or mobile device where code changes are relatively infrequent and traditional deployment techniques can be used. When continuous code updates are expected the cost of compiling, packaging, and distributing each change is prohibitive and not scalable.

Continuous Development

For large-scale deployments with continuously evolving functionality, a distributed deployment scheme must be utilized. For example, a new node is added to the system that is running an HOA platform daemon configured to join a HOA network and make itself available for serving data and/or running snippets. Distributed management facilities then decide which data and/or snippets to allocate to the daemon according to some high-level policy specification,
this could include data/snippets from any number of different completely independent applications defined in the policy. Updates to a snippet can then be pushed to daemons to replace older versions of the snippet.

Such a deployment mechanism can also be used during testing to place snippets and data on a dedicated subset of nodes in the system. In-situ integration testing in a production system may also be possible utilizing specialized sandboxes that segregate snippets and data being tested from the rest of the system. These testing sandboxes will allow snippets to receive diffused data from the production system but prevent them from diffusing data outside of the sandbox. Instead the diffused data is compared against expected diffusion patterns specified as part of the test configuration or observed in the production system.

Monitoring and Analysis

Distributed monitoring techniques can be integrated into the HOA platform to allow for scalable real-time logging and profiling of a running system. Such facilities are also required by autonomous computing services that can react to real-time metrics to optimize performance, proactively detect states that lead to failures, and recover from failures.

Case Study: Web-Based Gaming Startup

In a Web startup, system requirements are in a constant state of flux as the organization grows and responds to market feedback. A rapidly growing user base can also quickly
overwhelm their systems, requiring continual capacity expansion to handle the increasing load. These factors make a Web startup a good example to study how HOA can be used to improve scalability and evolvability in a real world scenario.

This study considers a startup creating a browser-based massively multiplayer online game. Within the game players will be able to navigate seamlessly within a virtual world populated with artist and player generated game content such as player avatars, terrain, buildings, etc. Such a game requires a sophisticated client-side UI utilizing modern web development techniques like HTML5 WebSockets or Comet to allow the server to push events and content to the UI in real-time. On the server side, a highly scalable HTTP server implementation is needed to handle large numbers of concurrent requests and real-time updates to client-side UIs. Game content is indexed using a spatial database to allow for fast retrieval based on current player locations within the virtual world.

The startup begins development with a handful of servers running a distributed HOA middleware platform that provides a number of development tools out of the box, like groupware, version control, issue tracking, automated testing, and system monitoring. All of these tools are themselves applications running on the middleware platform that can scale and evolve alongside the product.
The developers first create two spaces: one for game content and another for game logic. The game content space will be filled with spatial objects containing references to content. The HTTP server in the logic space will use these references to retrieve the actual content from a central storage location for distribution to clients, such as a third-party content distribution network or distributed file system managed by the startup. The reference could also point to player-generated content hosted by a third-party content creator or on a player’s own servers. The content space will be accessed from the logic space using multi-spatial snippets which instantiate themselves within the content space to represent the location and area-of-interest of players and retrieve content in proximity to them (depicted in Figure 4 by the edges between snippet instances in the logic and content spaces).
The developers then set about creating the first snippets in the game logic space to implement an initial prototype of the game server architecture. These snippets will implement the HTTP server, server-side presentation logic, and the player state machines.

The HTTP server will likely be an off-the-shelf snippet or a custom snippet utilizing a high performance HTTP server library available in the language the snippet is implemented in. The server snippet is responsible for managing session state, distributing local content to clients, and relaying UI events from clients to presentation snippets. The server may be executed on any number of the machines in the system depending on the underlying network topology. For the prototype implementation, only one machine is allocated to execute a single replica of the server snippet. As the system scales up to handle more users load balancing techniques can be used to distribute load across multiple systems running replicas of the server snippet.

Presentation snippets manage the server-side state of the UI. Client requests to update UI state or retrieve UI content are relayed to the presentation snippet that updates server-side state. The updated presentation state may trigger diffusion to other proximate snippets like the player state snippet that performs necessary operations and then diffuses a presentation state update back to the presentation snippet. This update is then processed to generate client UI update commands that are diffused to the server snippet to be transported to the client.

Player state snippets handle most of the actual game logic, such as player motion and interactions with other neighboring players in the content space. They instantiate themselves in the content space to represent the locations of players and are responsible for updating these
instances as players move throughout the virtual world. Player state snippets are also responsible for notifying presentation snippets about content and other players in proximity to a particular player as it moves through the content space. Player state updates sent from the presentation snippet include a player id that is associated to a particular instance of the player state snippet within the content space. These instances are modified as necessary to reflect the movement or changing area-of-interest for individual players. Player state could also be managed using dedicated player state snippets, each processing data diffused from the presentation layer for a particular player and associated with a single instance in the content space. Different types of players could also be implemented using specialized snippets to encapsulate the differing functionality.

After completing this initial testing of the prototype the developers begin work on adding two additional features: non-player characters and a billing system to manage player subscriptions and purchased content. Character snippets are implemented in much the same way as player snippets but instead use AI techniques to allow them to autonomously interact with other players, characters, and content. Like player snippets they instantiate themselves in the content space and interact with the presentation snippet to update client UI state.

Implementing the billing system requires a player authentication, authorization, and auditing snippet to manage player credentials, authorized content, and track account activity. The HTTP server uses the AAA snippet to authenticate users and associate a player id with a user session. The billing snippet interacts solely with the AAA snippet to add and remove players and
authorize access to purchased content. The billing snippet may implement a complete billing package providing its own HTTP UI for staff access, support for third-party payment processors, and any number of other features. It is more likely however that it will have to interface with an enterprise accounting system, but whichever method is chosen is inconsequential to the rest of the system.

With the prototype complete the developers move the system into production. In anticipation of rapid growth additional hardware is added to the system and the platform is configured to replicate snippets and state across all the machines. All machines are connected through a local area network but a subset of them will be connected to the Internet through a load balancer. The server snippet is configured to replicate over these load-balanced machines to maximize concurrency. The platform placement strategies will also replicate snippets in proximity to the server snippets, like the presentation and AAA snippets, onto the same machines running the server snippets or across the rest of the machines in the system depending on the performance characteristics of those snippets and the current load on the system. Remaining snippets like player and character state snippets will be similarly replicated across the machines in the system based on their proximity to the presentation snippets.

Soon after launching, an electrical issue causes a number of the machines to fail during peak load. The platform will detect the failure of these machines based on a lack of heartbeat messages or responses to diffusion messages. The platform will then begin recovering from the failure by replicating data and snippets that were placed on the failed machines onto the
remaining machines in the system. Because snippets are stateless they are simply restarted on these machine and begin to receive diffusion messages as though nothing occurred. Once the hardware failures are resolved affected machines are added back to the system to receive replicated state and snippets along diffusion messages load-balanced across snippet replicas.

After some time a large community of third party content developers has emerged and begin to demand greater access to the content space to build their own portions of the game world. These third party content creators establish their own HOA systems and register for access to the federated content space to begin receiving replicas of the content space from the original system. Placement strategies can be configured to prevent certain content from being replicated to or from these third party systems. Players exploring third-party controlled portions of the game world will require instances of their player state snippet to be placed in proximity to that content. The player state snippet may be executed exclusively on the original system or may be replicated to third-party systems to minimize network overhead associated with interacting with the content space. Over time other portions of the system may be federated in similar ways, for example the billing and AAA system may be federated to allow third party developers to charge for their content, or the HTTP server and presentation snippets may be replicated to improve load balancing.
CHAPTER 4: IMPLEMENTATION

Harmony.js is a prototype implementation of a non-distributed, single-process HOA platform supporting JavaScript snippets. It has been developed using Google's V8 JavaScript interpreter [48] and an asynchronous I/O and event-driven development framework for server side JavaScript built using V8 called Node.js [49]. The libev and libeio libraries used by Node.js for asynchronous I/O are extended to provide the event dispatching and I/O facilities needed in Harmony.js (asyncio in Figure 5). A simple document database is implemented using the Generalized Search Tree (GiST) library from UC-Berkeley [50] and BSON serialization library from MongoDB [25] (asyncdex in Figure 5).

Figure 5: Harmony.js platform components.
JavaScript Snippets

JavaScript was chosen as it has interpreters implementing native sandboxing facilities, tremendously simplifying the complexity of the implementation. It has a long history of use in web browsers requiring isolated execution of untrustworthy code and more recently, several high performance JavaScript interpreters have been implemented to support modern client-side web applications. The open source Google V8 interpreter was chosen for this prototype for its performance and relative ease of use. However, it does not provide any form of standard library to JavaScript snippets, useful for sandboxes in a web browser, but not so useful for general-purpose application development.

Node.js is a JavaScript application server that uses V8 and provides a set of libraries providing asynchronous access to files, sockets, module loading, HTTP clients and servers, and other facilities needed for implementing JavaScript server applications. Node.js applications have been shown to be highly scalable and capable of supporting large numbers of concurrent connections thank to its asynchronous nature. Portions of Node.js were adapted for use as a standard library for JavaScript snippets.

In Harmony.js, each snippet is executed in its own sandbox that encapsulates an isolated V8 execution context and any state associated with Node.js or the Harmony.js Snippet API (Figure 5). When a sandbox is created, a new V8 execution context is created and activated to initialize the Node.js library and the Snippet API before executing the code associated with a snippet. Sandboxes also hide the underlying language implementation by exposing JavaScript
code within the V8 execution context through a generic interface used by Harmony.js (to call a JavaScript event handler defined in a snippet for example). In the future different types of sandboxes will be implemented to support different languages and isolation models.

Asynchronous Execution

The libev and libeio libraries provide asynchronous file and socket I/O (simulated using background threads when necessary). These libraries are also used to support the asynchronous database operations and event dispatching required in Harmony.js (asyncio in Figure 5).

Event dispatching between snippets is implemented using a simple event queue which is processed every iteration of the libev main loop. Events in the queue store the event ID and any data associated with the event. Publishing an event is as simple as inserting it into the queue. Dispatching an event to registered event handlers in different snippets requires a mapping from event IDs to a list of V8 contexts associated with the snippets. When a snippet subscribes to an event through the Snippet API, the mapping is updated to include the context associated with the snippet (if not already present). When the event queue is processed, the list of contexts mapped to an event ID is retrieved, each in turn is activated, and a JavaScript function in the context is called to invoke any functions registered in the snippet to handle the event.
Figure 6: Event dispatching example.

Figure 6 depicts the process of dispatching a diffusion event from one snippet instance to another. Instance1 calls a `diffuse` function defined by the Snippet API to diffuse data to its neighbors in a Space. The Snippet API creates a diffusion event containing the data and a reference to the snippet, instance, and context containing them. It then calls an internal `publish` function which is a JavaScript wrapper around a C++ method of the Event Dispatcher that queries the Spatial Index of the Space containing Instance1 for neighbors within diffusion range, in this case Instance2. The diffusion event is then updated with a list of references to the neighboring instances and associated snippets. It is then enqueued for later dispatching when the Event Dispatcher is called from the event loop, at which point, the event is dequeued and for each referenced snippet the event is passed to the event dispatching method of the snippet's sandbox. It activates the associated V8 execution context and calls a JavaScript event dispatching function that invokes any registered diffusion event handlers for Instance2.
Indexing

The primary bottleneck of an HOA platform is the database, as it must be frequently queried to diffuse events and access persistent snippet state. Choosing indices well adapted to the data and access patterns is crucial to ensuring optimal performance. However, no matter which index is chosen it will be I/O bound and fundamentally limited by the performance of the underlying storage model. Caching frequently accessed data or relying on non-persistent in-memory indices is the only way to avoid these costly I/O operations.

For example, diffusion query results can be cached to improve event-dispatching performance for infrequently moving snippets. Unfortunately, the spatial nature of these queries requires costly searches to locate cache entries to invalidate after a snippet is moved. Avoiding them requires that the entire cache be invalidated, rendering the cache ineffective for frequently moving snippets. These cases can usually afford to compromise persistence for performance by using in-memory indexes like those used in gaming and simulations [51–53]. Harmony.js caches diffusion query results using a Least Recently Used cache to reduce the significant cost of accessing GiST indexes, though other cache algorithms will be explored in future work.

The present implementation of Harmony.js uses the B+Tree and R*Trees provided by the Generalized Search Tree (GiST) from UC-Berkeley [50], [54], [55]. GiST is an abstraction framework for tree-based indices that uses dependency injection to provide the general functionality needed by most indices (such as traversal, storage operations, etc.) while delegating to specialized implementations for specific types of indices. It was chosen primarily because it
supports both spatial and non-spatial indices using the same storage model, greatly simplifying the database implementation. This has the advantage of simplifying the development of specialized indices and allowing new indices to be easily integrated into an existing application. Another advantage is that existing index implementations can use new storage models without any modifications.

However, the UC-Berkeley implementation of GiST lacks thread-safety and lock-free concurrency, which precludes any form of asynchronous processing without significant modifications. To ensure non-blocking execution all operations on GiST indices are performed from the libeio thread pool and protected by a global lock to serialize operations. While this execution model is sufficient for the current prototype, in the future a more concurrent index implementation will be necessary. This could be achieved by simply implementing granular locking, but this will not eliminate the cost of blocking in background threads. The ideal solution would support asynchronous operations natively using lock-free indexes with asynchronous I/O [29–31].

**Document Database**

The database design in Harmony.js is inspired by NoSQL systems like MongoDB and CouchDB [25], [26]. It provides document databases using simple key/value indices to associate keys with BSON documents, a binary version of JSON developed for MongoDB. Support for multiple document collections that consist of a set of documents and a number of indices over
them was also inspired by MongoDB. This design allows a special system collection to be used
to store database configuration for collections and related indexes and Harmony.js configuration
details for snippets, spaces, and instances. It also allows collections to be associated with each
space to store and index spatial objects contained in the space (see Figure 7).

Figure 7: Database design.

Snippet configurations specify the JavaScript code to execute in the form of either a file
path or code stored directly in the configuration document. Space configurations specify their
dimensions, extents, and a collection storing documents representing spatial objects like diffused
data or snippet instances along with spatial indices to look up spatial objects within diffusion
range of a snippet. Instance documents stored in a space collection specify the snippet which
they instantiate, the space in which it is instantiated, and the spatial properties of the instance
which may be indexed.
When the Harmony.js server is started, Snippet and Space documents are retrieved to create sandboxes and open space collections. Each sandbox is created with references to all of its instances and the spaces that contain them. Snippet code is executed within each sandbox to initialize the snippet and register asynchronous event handlers. The event loop is then started to begin dispatching events generated while initializing the snippets.
CHAPTER 5: EVALUATION

Figure 8: Snippet chains

To determine the baseline performance characteristics of this early prototype of Harmony.js a demonstration application has been created to gauge various performance characteristics. This application utilizes an HTTP server snippet that diffuses requests into a space and waits for responses from other snippets. A “chain” of request handling snippets in proximity to the server snippet waits to receive requests that are relayed between “links” along the chain (see Figure 8). Once the request has reached the end of the chain a response is relayed back towards the server snippet to generate the HTTP response. To properly evaluate the performance of the event dispatcher link snippets must be implemented as singleton snippets to ensure that costly inter-sandbox diffusion occurs rather than less costly intra-sandbox diffusion between instances running in the same sandbox.
These “snippet chains” provide a simple model demonstrating the type of data processing pipelines that will be common in HOA applications. Individual links in the chain may perform any number of operations when handling a diffused event, such as accessing data stored in a database, processing data in some way, performing network or file I/O, etc. These links may be arranged to form complex graph-like pipelines with multiple interconnected data sources and sinks forming “fields” of data flowing through the links. No matter the complexity of the arrangement its performance will be bounded by the performance of a simple linear snippet chain, making them a suitable model for analyzing the performance characteristics of the Harmony.js platform.

To evaluate the performance of the system the ApacheBench web server-benchmarking tool is used to gather performance statistics over a range of concurrency levels [56]. These benchmarks are performed on the same machine running Harmony.js to avoid benchmarking the performance of the underlying network. As such they do not reflect real world performance where network latency can significantly impact performance. For a given concurrency level ApacheBench launches parallel request generators that repeatedly issue requests and block to await responses. The maximum number of concurrent requests is limited by the overhead incurred by multi-threading and system constraints on the number of open sockets and available ports. This is not a problem for testing less efficient servers but generating enough concurrent requests to saturate a Harmony.js server can easily exhaust system resources. Due to these and other limitations of ApacheBench the results are highly subjective and are only suitable for comparing the relative performance of Harmony.js across experiments.
Varying Chain Length

This experiment will evaluate the event dispatcher performance by using a snippet chain in which link snippets do not access the data in the space to keep the non-event related database load constant. Each HTTP request received by the server snippet generates an event that is diffused to the first link snippet in the chain. Each event is assigned a Time-To-Live that determines the total number of “hops” along the chain that the event will take. Each link snippet that receives an event will append some data to the event, decrement the TTL, and diffuse the event to the next link in the chain. When the TTL reaches zero the event is repeatedly diffused back along the snippet chain towards the server snippet. For a given TTL the event will take twice as many “hops” along the event chain. Four variations of this experiment are performed with TTL values of zero, one, four, and eight.

Varying Link Cost

In this experiment the overall database performance will be evaluated using a fixed TTL for each request and a snippet chain in which each link accesses the database a varying number of times. At each link a document containing a 512-byte field is persistently diffused into the space during the initialization of the link snippets (a larger size would have been used but a bug in the GiST implementation limited the maximum size of the documents). For each event received at a link, whether coming from or going to the server snippet, all the documents in proximity to the snippet will be retrieved. Because each link is in proximity to two other
snippets, a total of three documents will be retrieved for each “observation” made from a link. However, the link proximate to the server snippet will only receive a total of two since it only has one neighboring link with a diffused document.

Four variations of this experiment will be performed using zero, one, two, and four observations per link. The chain length will be fixed at four resulting in a total of eight hops per HTTP request, but only seven links will be traversed, and all but one, which receives the event with TTL of zero, will perform their observations twice. The link next to the server snippet will be traversed twice but will only observe two documents at a time; all others will observe three documents each. In each variation a total of 0, 19, 38, and 76 documents will be retrieved during the processing of each HTTP request. The 512-byte fields of each retrieved document will be concatenated and returned in the HTTP response, leading to approximately 0kB, 10kB, 20kB, and 40kB of data retrieved from the database and returned to the client per request.

Experimental Setup

For all experiments, unless otherwise noted, ApacheBench is executed for 60 seconds with the number of concurrent requests varying from 1 to 1000 in increments of 100. For each concurrency level results are averaged over five runs and the Harmony.js server is restarted before each run. The maximum concurrency level was set at 1000 in the varying chain length experiment due to failures occurring in ApacheBench as it exhausted system resources at higher concurrency levels. For the link cost experiments the maximum level was set to 100 due to the
high latency that was causing ApacheBench to wait for request to complete for much longer than the requested 60 seconds.

All experiments were performed on a system running Ubuntu 10.04 with a quad-core 2.33GHz Intel® Q8200 processor and 8GB of memory. Harmony.js does not support multi-processing directly however background threads used to simulate asynchronous operations may have been assigned to different processors. Caching was also disabled to better gauge the performance cost of the database operations and avoid evaluating the caching algorithm.
CHAPTER 6: RESULTS

Interpreting ApacheBench results requires understanding the relationship between the average request rate and total response time for a batch of concurrent requests. The total response time usually increases in proportion to the number of concurrent requests due to requests being multiplexed across available processors. For example, if on a single process system 1000 concurrent requests are processed at a rate of 100 requests per second (0.01 second per request) then the total response time is 10 seconds per 1000 concurrent requests. With a two-processor system and assuming ideal scale-up the request rate should double resulting in half the total response time.

For the total response time to remain constant under increasing concurrency, requests must be processed in parallel to increase the overall request rate to match the concurrency level. Once the concurrency level becomes greater than the number of available processors the request rate becomes constant and the total response time begins to increase in proportion to the concurrency level as more requests are multiplexed across the same number of processors. At a certain concurrency level the request rate begins to fall and the total response time increases exponentially as more time is spent multiplexing requests and waiting for system resources than processing requests.
In the following results where the performance of each variation of chain length or link cost are compared, the per-request response time (the inverse of the request rate) is shown rather than the total response time since it can be determined by multiplying the per-request response time by the concurrency level.

**Varying Chain Length**

A comparison of the performance for several TTL variations is shown in Figure 9. The response time remains fairly constant with increasing concurrency for each variation of the TTL indicating that the platform scales well up to 1000 concurrent requests. What is also interesting is that the response time is almost directly proportional to the number of hops along the snippet chain, indicating an almost constant cost per hop.

![Comparing Performance Under Varying Chain Length](image)

Figure 9: Comparing performance with varying snippet chain length.
Detailed results from each of the TTL variations are shown in Figure 10. While the max request rate appears to remain relatively constant across all variations there is a noticeable decrease in the average and minimum request rate as the TTL increases. The degree of variance also seems to increase with larger TTL, which is likely due to the limited number of runs the results are averaged over.

Figure 10: Detailed performance with varying snippet chain length.

Varying Link Cost

The resulting performance under varying link costs is shown in Figure 11. As link cost increases there appears to be a proportional increase in the response time. As can be seen, the
response time for two observations per link is double the response time for one observation per link. Between two and four observations per link there appears to be greater than two-fold increase. This indicates there may be a greater than proportional increase in the cost of accessing the database as the total number of accesses increases. There is also a significant spike in response time at higher concurrency levels with four observations per link. This is likely due to rounding error since the request rates reported by ApacheBench are recorded as whole numbers and with four observations per link the request rate is very low.

![Comparing Performance Under Varying Link Cost](image)

**Figure 11:** Comparing performance with varying link cost

Figure 12 shows the detailed results for each variation of link cost. The request rates fall dramatically with the increasing number of observations per link. The high degree of variance is partially due to request rates being recorded as whole numbers but may also be due to the high cost of database access with the current implementation.
The results from the evaluation of the snippet chain demonstration application have shown Harmony.js to scale relatively well considering the inefficiency of the implementation. In all cases the average request rate appears to remain relatively constant over all concurrency levels, although because it is a single-process implementation the latency increases proportional to the concurrency level.

Varying the snippet chain length has shown the per hop overhead to be relatively constant. Considering that each hop involves serializing a JavaScript object to JSON and then to
BSON and deserializing it back into a JavaScript, the per hop cost can likely be significantly reduced by optimizing this process. There is also some room for improvement in the performance of the event dispatcher itself, by optimizing the event queues and integrating intra-sandbox event dispatching directly into the libev event loop.

The results from the link cost experiments show that there is a significant performance cost associated with accessing the database, which is not unexpected considering that all database accesses are performed in sequence without any concurrency. Clearly an indexing implementation with a better concurrency model is necessary and should significantly improve performance. The increased concurrency will also improve the event dispatcher performance, which depends on spatial queries to determine snippet instances to receive diffusion events.
CHAPTER 7: CONCLUSION

Harmony Oriented Architecture is a novel architectural paradigm for implementing scalable and evolvable systems based on the concepts of Harmony Oriented Programming. It is motivated by the need to remove humans from the design process to allow self-designing Ultra Large Scale systems to emerge. A middleware platform for constructing distributed systems using HOA is proposed which can serve as a foundation for implementing a large-scale self-designing system. An early non-distributed prototype of such a platform is demonstrated and evaluated to characterize its performance and guide for development.

Harmony Orientation rejects the information hiding and encapsulation of Object Orientation that leads to brittle and overly complex relationships amongst objects in a large-scale application. Instead, state exposure and functional isolation are used to ensure decoupling between objects and improve evolvability. A spatial abstraction is used to mediate the interaction between “snippets” of isolated code through a process of data diffusion. Rather than interacting directly through specified interfaces, snippets are forced to interact indirectly by diffusing data into the space for proximate snippets to observe. The arrangement of snippets within the abstraction determines the overall “flow” of diffused data and by manipulating this arrangement the overall functionality of the application can be altered dramatically without any modification to the snippets themselves.
Harmony Oriented Architecture applies the principles of Harmony Orientation to the architecture of large-scale systems. It is proposed as an alternative to Object Oriented architectural techniques like Service Oriented Architecture or Component Based Architecture that utilize graph-like structures to represent interfaces and relationships between application components. Snippets in HOA can be implemented in potentially any language and are run in complete isolation. They may only interact by diffusing data to proximate snippets represented as spatial objects stored in a distributed spatial database. This is similar to how tuple spaces are used in Space Based Architecture to allow application components to interact indirectly, but SBA does not store the components or represent their relationships using tuple spaces.

Snippets provide a shared-nothing, message passing concurrency model that enables applications to be highly scalable and evolvable. Storing snippets and their state within a spatial database improves the evolvability of applications by allowing snippets and their relationships to be easily modified by both humans and machines alike. Snippets can be readily replaced by new versions, rearranged to alter functionality, replicated across machines, and restarted or relocated to adapt to failures and changing usage patterns, all with just simple database operations without any need to modify to the snippets themselves. Representing snippet relationships using spatial arrangements also facilitates the autonomous evolutionary processes necessary for a self-designing system that would otherwise be too complex using the graph-like relationships available in Object-oriented paradigms.
The initial prototype implements a non-distributed single-process middleware platform to demonstrate the feasibility of the concepts HOA. Considering the inefficiency of this early stage prototype it performs remarkably well, at least in regards to event dispatching between snippets, which appears to have a constant cost. The results also highlight the inefficiency of the non-concurrent index implementation and demonstrate the need for further improvement.

Contributions

The most significant contribution of this work is the extension of the concepts introduced in HOP to the architecture of large-scale distributed systems. The high degree of decoupling and isolation provided by snippets enables highly scalable applications to be built with relative ease. The spatial abstraction used to represent the relationships between snippets provides a general-purpose “programmable” architecture that exposes high-level architectural features to applications components enabling the architectural reflection and modification necessary for self-design.

Future Research

This thesis represents the first step in exploring the use of Harmony Oriented Architecture for building ultra large-scale systems. There is still a significant amount of research and development necessary to implement a production quality HOA platform. What follows is a brief overview of current plans for future research.
Improving Concurrency

The current GiST implementation is an unnecessarily costly bottleneck affecting the performance of both database accesses and event dispatching between snippets. There are a number of highly concurrent indexing options available to consider, including extensions to GiST which improve concurrency for tree based indices [29], [30], [57–60]. The greater degree of concurrency will also require transaction-processing facilities to allow concurrent database operations to be performed safely. Because snippets execution is essentially managed by the database it has a much insight into application behavior that could help optimize transaction processing. Another limitation of the current prototype is the single-process execution model that precludes implementing CPU-bound snippets without blocking the event loop for long periods of time. Executing snippets across a number of processes will not only improve concurrency and allow CPU-bound snippets to be executed safely but is also the first step towards supporting distributed snippet execution.

Distributed Architecture

There is extensive research available on a variety of distributed architectures for database and application servers that will be explored for use in Harmony.js. The current plan is to implement a Peer-to-Peer networking platform to implement a scalable distributed spatial database and event-dispatcher. Other options like grid-based or client-server models may also be considered.
Another interesting concept that will be explored is using the spatial abstraction offered in HOA to represent aspects of the distributed system. This can allow for a variety of system properties to be exposed and manipulated by snippets, including the geographic locations of nodes, their network connectivity, or even hardware capabilities (to allow hardware-aware snippets to be placed appropriately for such things as GPU-optimized snippets or snippets exposing various types of hardware like screens, storage, telephony equipment, etc.).

**Commensalistic Architecture**

Full support for commensalistic architecture will require extensions to the platform to support specialized host snippets with access to a privileged API for manipulating symbiont snippet internal state for debugging and recovery purposes (such as restarting a failed snippet). Various implementations of host and symbiont snippets will be evaluated to determine their effectiveness and performance characteristics for different types of failures or abnormal behavior.

**Evolutionary Architecture**

The ultimate goal of this research is to provide a platform for real-world in-situ application of evolutionary computing techniques in large-scale systems as alluded to in [1], [2]. The spatial computing approach that underlies HOA can allow for significant variation in program behavior without any modifications to the snippets. One approach being considered is to use a collection of simple manually implemented snippets placed in a space in which their
arrangement is evolved to optimize various characteristics of their overall behavior. Additionally evolutionary programming techniques could also be used to evolve individual snippets directly, potentially allowing for completely automated evolution of large-scale systems.

Evolved snippet arrangements are similar in concept to the evolved arrangements of multi-cellular organisms. Such snippet ensembles could implement self-organizing behaviors by applying research in artificial embryogenesis and morphogenesis to allow these “organisms” to grow from “germ” snippets or alter their spatial arrangement to adapt to changes in the environment or required functionality.

The use of “intelligent” snippets capable of observing and reasoning about data diffused by neighboring snippets them can be used to implement phenotropic behavior. Furthermore, the use of these intelligent snippets in ensembles could allow for “introspection” into an ensemble and possibly enable directed evolution of its behavior. These techniques can also be applied to commensalistic snippets to allow hosts to be evolved with intelligent behavior to better manage symbionts and provide greater autonomy and reliability.

Social Dynamics in Large Scale Systems

One of the primary limitations in any computer system is the human factor. This is even more of a concern in ultra large-scale systems since most system constraints are human imposed and they have little or no bearing on the core functionality required for implementing the system. For example, access controls in a federated system can be prone to misconfiguration due to
improper management and general inefficiencies in large organizations that can limit manageability and lead to failures. For HOA to successfully be deployed in real world large-scale systems it is crucial that mechanisms be in place to meet these human requirements while also managing the humans using the system to limit their effects.

The development lifecycle services described earlier form one component of a larger social management facility. Social networking and groupware services can be included to facilitate communication and collaboration within and between organizations managing a HOA system. Authentication, authorization, and accounting (AAA) services are crucial to securing the system against unauthorized usage from malicious users. Monitoring and event analysis techniques can be used to identify patterns of malicious user or application behavior and pre-failure states for proactive failure mitigation. Large-scale domain automation can also be achieved using these same techniques to analyze patterns in the domain model and adapt it changing circumstances.
REFERENCES


81


