Hatch: Self-Distributing Systems for Data Centers

Roberto Rodrigues-Filho[a], Barry Porter[b]

[a]Institute of Informatics, Federal University of Goiás, Goiânia-GO, Brazil
[b]School of Computing and Communications, Lancaster University, Lancaster, UK

ARTICLE INFO

Keywords:
Self-distributing Systems
Emergent Systems
Autonomic Computing

ABSTRACT

Designing and maintaining distributed systems remains highly challenging: there is a high-dimensional design space of potential ways to distribute a system's sub-components over a large-scale infrastructure; and the deployment environment for a system tends to change in unforeseen ways over time. For engineers, this is a complex prediction problem to gauge which distributed design may best suit a given environment. We present the concept of *self-distributing systems*, in which *any local system* built using our framework can learn, at runtime, the most appropriate distributed design given its perceived operating conditions. Our concept abstracts distribution of a system's sub-components to a list of simple actions in a reward matrix of distributed design alternatives to be used by reinforcement learning algorithms. By doing this, we enable software to experiment, in a live production environment, with different ways in which to distribute its software modules by placing them in different hosts throughout the system's infrastructure. We implement this concept in a framework we call *Hatch*, which has three major elements: (i) a transparent and generalized RPC layer that supports seamless relocation of any local component to a remote host during execution; (ii) a set of primitives, including relocation, replication and sharding, from which to create an action/reward matrix of possible distributed designs of a system; and (iii) a decentralized reinforcement learning approach to converge towards more optimal designs in real time. Using an example of a self-distributing web-serving infrastructure, Hatch is able to autonomously select the most suitable distributed design from among ≈700,000 alternatives in about 5 minutes.

1. Introduction

Designing, deploying and maintaining distributed systems remains one of the most challenging tasks in computing. This complexity leads to a well-worn path of *design, deploy, analyze, redesign* to get the most out of large-scale operations. The sources of this complexity are multi-faceted, encompassing the size of a code base; the dynamics of the deployment environment; complex interaction effects between multiple sub-systems; and the constant evolution of the underlying hardware on which these systems operate.

This complexity is acutely present in the software operations of data centers for large-scale web infrastructures such as those of Facebook and Google, which have well-known challenges in both the placement and design of individual sub-systems which are hard to predict prior to real deployment [45], and are affected on a continuous basis by the specific mixture of shifting request traffic patterns being experienced at any given time [3]. Today this complexity is only understood via painstaking and largely manual analysis feeding into manual remediation workflows, with a range of tools that attempt to assist in this analysis [40, 50, 22]. Designing the right distributed architectures therefore remains hard; analysis of those systems in their production environment is very challenging; and this analysis and remediation actions are a constant effort as workload patterns change over time.

We present a radically different approach to tackling distributed systems complexity, which closes both ends of the analysis-redesign loop: Hatch takes any local system, which is built from set of fine-grained building blocks such as stream parsers and hash tables, and implements the ability to seamlessly and safely distribute any building block to a remote host, while the system is running, without any of those blocks being explicitly designed for distribution. We build a set of distribution styles on top of this capability, including relocation, replication for both stateless and stateful building blocks, and sharding (where the state of a component is distributed among the component's replicas). Hatch then uses real-time reinforcement learning to explore potential distributed system designs from within the resulting search space. Our approach is generic to any modular system, and allows engineers to focus only on the local design of a system, fully delegating its distributed design to autonomous processes. Hatch requires no training, learning everything in real-time from the actual environment being experienced by the system, and every transition to a new system design is seamless in missing zero traffic. To realize this vision, our approach comprises three major elements:

1. We implement a generalized mechanism to safely and seamlessly *self-distribute* any building block while a system is running, to transform a local system into a distributed one. This mechanism is mainly realized by a transparent and generalized Remote Procedure Call (RPC) layer, which enables runtime relocation of any local component to any remote host in the data center, automatically transforming all local interaction with the relocated component to RPC.

2. We abstract complex distributed design choices currently made by engineers at design time to a simple action/reward matrix to be explored at runtime by reinforcement learning algorithms. This allows us to automate decisions on which building block variants to use, where to place them, and how

ORCID(s): 0000-0002-3323-0246 (R. Rodrigues-Filho); 0000-0001-8376-736X (B. Porter)

and when to replicate them, using simple non-disruptive distribution actions to create different distributed designs, while observing the impact of these decisions on the live system.

3. We present a distributed machine learning approach to navigate the search space of primitive actions and locate increasingly optimal designs of a data center web-serving infrastructure. This learning approach fuses both local optimizations on each host (such as choice of caching algorithm or request bursting approach) with distributed optimizations which relocate or replicate selected sub-behaviors.

Our research draws inspiration from previous notable work in the literature and from our own past project RE^x[33] where the application of machine learning techniques are central in designing and configuring systems, both at system and application levels such as [49], [38] and [33]. We innovate by looking exclusively at the distribution aspect of systems from the application level, focusing on addressing the complex challenge of deciding which distributed design will better suit different operating conditions. A key enabler of this technique is to use small building blocks of software, similar to [38] and extended from [33], which gives more flexibility in how to compose the resulting system, making it more versatile in how the system performs under diverse conditions.

The result of our work is *self-distributing* systems, in which the placement and detailed design of each part of the overall system emerges at runtime as a direct result of the operating environment in which the system currently exists. We evaluate our approach by building a web-serving infrastructure in Hatch and subjecting it to a range of different workloads. We demonstrate how zero-training reinforcement learning can rapidly locate different local and distributed designs based on real-time perception and learning of how each workload affects the system. Our work is a step towards significantly reducing the complexity of building distributed systems, by having the system learn for itself how to assemble, deploy and distribute. Our implementation, with instructions on how to reproduce all of our experiments, is available online[1].

This paper is organized as follows: we survey related work in Sec. 2. Sec. 3 provides background on the technology we use to build Hatch. We discuss our approach in Sec. 4, present our evaluation results in Sec. 6, and conclude in Sec. 8.

2. Related work

We survey related work in two main themes: general distributed computing technologies, which seek to bridge local / distributed concepts using various approaches; and research around automated analytics and learning in data centers. In both areas, Hatch is unique in making complex real-time design choices on how to architect and deploy a distributed system. We note that a range of existing research using the term "self-distributing", such as [48, 18, 19, 26], does not fully implement the concept of self-distribution the same way, relying heavily on developers to explicitly define distribution directives to be performed at runtime. Our approach, by contrast, considers fully local systems built to run on a single host, and with no information on the system's operating environment, infrastructure nor further development directives, is able to autonomously relocate sub-elements of the system's composition to remote hosts. The novelty of Hatch lies in our complete transparent and generalized RPC layer which enables self-distribution of local components with no third-party software or infrastructure aid (e.g. containers, virtual machines, cloud), and the abstraction of complex distributed designs choices to an action/reward matrix to be used by reinforcement learning algorithms during execution time.

2.1. Distributed computing technologies

The idea of blending local and distributed systems under an object model is explored in the seminal 1994 paper by Waldo et al [47]. The authors identify challenges that may arise in doing this, particularly on error propagation for partial failures where a lack of remote error codes on APIs designed to be local creates a dilemma on how to propagate those errors. This analysis lead the authors to conclude that all remote objects should be behind explicitly remote interfaces, treated separately to local objects by the programmer. This in turn led to the design of systems like RMI [25], CORBA [4], and COMPSs [41], which define coarse-grained interfaces to explicitly describe remote services, with associated error handling being placed at the feet of developers. The drawback of these models is that human engineers must decide *which parts* of a system to distribute and *how*, and to revisit this as systems evolve over time. Instead, we propose seamless continuity between local and distributed systems, creating a unified real-time search space for online reinforcement learning to navigate. To support this, we use an approach to fault-tolerance not originally considered by Waldo et al, in which we only use transparent distribution of local code if we can also silently recover from failures.

The closest research to our own in local/distributed continuity is the recent work on disaggregated data centers at a hardware level [16], and LegoOS as a software solution to leverage their capabilities [38]. LegoOS divides a traditional operating system into separate 'monitors' for major elements of the OS (processing, memory, and storage). These monitors are then organized into a distributed system across a data center by the LegoOS management process – which also transparently handles inter-host communication as if it were all local. When a process is executed it is assigned to a specific processing monitor and all threads launched by that process reside in the same monitor; memory allocation is abstracted so that individual memory monitors can be spawned on new hosts to give the abstraction of continuous memory across hosts, with the same approach taken for storage. Our research takes a very different direction to local/distributed continuity, leveraging a strong component model to gain seamless distribution of *parts of a process*, rather than parts of an OS which map to hardware con-

[1] Access our project's code at: http://projectdana.com/fgcs2021rodrigues

cepts. This provides a richer set of deployment options to divide/replicate parts of a process across many hosts, including deciding which code to colocate.

Beyond generalized approaches to distributed system building, there is an increasing set of specialized solutions available to try to bring distributed closer to local. This includes hardware offerings such as RDMA and InfiniBand, for which parts of a software system can be specially refactored to gain performance (e.g., [6][13][14]); and software libraries such as the recent eRPC protocol [23] for rapid remote procedure calls – which offer an asynchronous, optimistic execution paradigm and optimize deeply for the common case in modern data center hardware. The core benefit of our generalized approach is that learning agents can freely decide on where to place each element of a system, depending on how that system actually performs in its current environment; however, many of these specialized approaches are complementary to our own, and could be integrated on a per-case basis using custom proxies.

2.2. Analytics and learning in data centers

Analysis of traffic and performance for data center systems has long been a focus of systems researchers, and is ever more prevalent with the increasing role of data centers in modern services. There is also a renewed focus on how machine learning advances can be used to aid in decision making.

In systems analysis, Veeraraghavan et al. [45] present an approach to stress-testing production systems at Facebook, by subjecting live data centers to very high workloads and inferring the root cause of problems from sparse monitoring. Following initial data collection, however, analysis is performed manually and then development teams are tasked with finding solutions. Similar tool support has been developed to analyze critical paths in dataflow applications such as Spark and TensorFlow, again to assist in offline design refactoring [20], and in automated profiling of cause/effect relationships in data center performance to flag issues to engineers [50]. While this research addresses one side of distributed systems complexity, our research takes a key step towards automating the *end-to-end* process via joined-up autonomous *monitoring* and *live redesign* of a distributed system.

In complementary work, Ardelean et al. [3] report on the challenges of analyzing performance of the Gmail infrastructure, and of deploying effective strategies to enhance this performance. The authors observe that the specific mixture of user requests significantly affects performance in different ways over time, making a single point-analysis of limited value; they also observe that performing offline experiments of potential fixes is of limited value due to the difficulty of replicating the same load characteristics at scale, and so online approaches are needed to gain actionable insights. These observations motivate our work in pushing design decisions into the live system based on real-time observations of system and user behavior to draw correlations between the two.

In the application of machine learning, a range of recent research has aimed to gain automation in data center optimization, from deep learning to control cooling levels [15], to optimizing Apache parameters using control theory [11]. Particular examples for software optimization include Pytheas [21], Optimus [31], and CherryPick [2]. Pytheas uses explore/exploit reinforcement learning mechanics in quality-of-experience tuning for video streaming, in which CDN routing decisions are given to a learning-based process to determine which CDN each video streaming request should be routed to. Optimus, meanwhile, uses online fitting to predict how long deep learning tasks will take to optimize scheduling decisions; and CherryPick uses a Bayesian optimization approach to choose ideal VM configurations for cloud applications, including instance types and cluster sizes. All of this research targets specific sub-concerns of existing systems, however; Hatch offers far more fundamental automation in the *generalized design* of a distributed system, by taking a local system and learning how best to distribute, replicate, or shard each element of its behavior and state across a hosting infrastructure.

3. Background

Our self-distributing approach targets object-oriented systems in general. The core idea is to relocate local running objects to remote hosts, with a transparent and generalized RPC layer to support this auto-distribution. This concept could potentially be implemented to an extent in a range of well-known languages (C++, Java), but their design would prevent a fully generalized solution – particularly as they lack generalized code hot-swapping, explicit dependency injection, and they allow writable shared memory between objects. Due to these constraints, we instead chose the Dana programming language [32] to implement Hatch which offers the potential for a truly generalized self-distribution approach for any program.

Dana is a component-oriented programming language with an emphasis on deep generality and safe, fast hot-swapping of behavior in microseconds. Everything in Dana is a hot-swappable component, from socket implementations to graphical widgets (in contrast to classic application-level component models which limit generality [7, 9, 30]). Hot-swapping facilitates the implementation of Hatch because it allows seamless runtime replacement of local running objects with proxies, which serves as local references of relocated objects, assisting in the implementation of our transparent RPC layer.

Dana is based on strongly-encapsulated software components, where each component must declare *provided* and *required* interfaces. An external management system can then decide how to wire up each required interface of one component to a selected (compatible) provided interface of another. This feature supports detailed dependency analysis of a whole system which helps reason about the collection of logic to relocate when moving a single object between hosts.

Dana implements this paradigm for full-stack systems software, for which it features an object model layered within

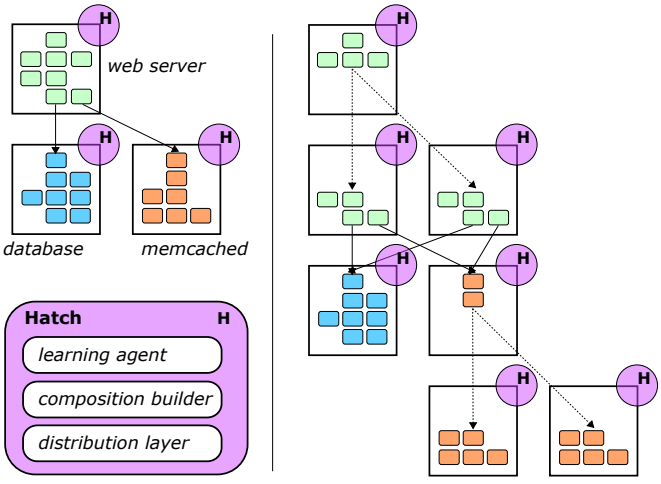


Figure 1: Multiple self-distributing systems running in a data center, each accompanied by a Hatch instance 'H' which observes performance in the current deployment conditions. Hatch can distribute local building blocks across hosts to learn which design alternatives best suit the current scenario.

its component paradigm together with a separate type hierarchy to represent pure data instances / arrays. Objects, data instances and arrays in Dana can all be passed by reference, such that Hatch must transparently accommodate reference graphs across auto-distributed components while maintaining transparent fault-tolerance in the distribution of code which was not explicitly designed to be spread across a network. A key benefit to the objectives of Hatch is that data instances and arrays in Dana are read-only at a callee, such that there is no shared writable memory, and object state can only be accessed via function calls, each of which make automated distribution easier.

Finally, Dana provides soundness properties for runtime adaptation in any program written in the language [34]. This means that the process of replacing components locally is completely safe and never leads the system to an inconsistent state. Porter and Rodrigues-Filho [34] demonstrate Dana's adaptation soundness when component hot-swap is carried out locally. Also, Porter [32] provides an in-depth view of Dana component-based model and how it supports seamless adaptation. While Dana is therefore a particularly well-suited platform for Hatch, its core concepts could be applied in more constrained ways to systems written in other languages with other component-based models [7, 9, 30].

4. Approach

In our previous project we built RE^x [33, 36], a framework that leverages the use of lightweight-component based technology [7, 9, 30] in tandem with multi-armed bandit algorithms [5, 42] to spontaneously compose software systems to better match (often unexpected) changes in their operating environment. RE^x is the first framework that provides full abstraction for reinforcement learning algorithms to learn how to best compose local software systems at runtime and successfully shown how machine learning algorithms can autonomously lead the designing process of software systems, supporting timely and accurate actions to deal with changes in the environment.

RE^x operates by composing software systems using small components (i.e. tiny fragments of software behaviour), and by replacing a component variant to another, RE^x allows the system to provide the same functionalities in a slightly different manner, impacting only some non-functional aspects of the system. The fact that each composition is functionally equivalent would be established by applying the same test suite to each composition at development time. Engineers can use RE^x to define goals (e.g. improve performance) and through the provision of metrics (e.g. average response time) collected from the running system, RE^x uses multi-armed bandit algorithms (e.g. Thompson sampling, and UCB1) to learn which local software composition yields the maximum satisfaction of the goal.

In this paper, we expand on RE^x's ideas to explore an entire new and highly challenging dimension of systems' design: *distribution*. Using the same philosophy we built Hatch, a framework that abstracts distributed system's design compositions to be used as actions for reinforcement learning algorithms. The main idea is to autonomously explore and learn distinct distributed software design at runtime to find the best system's composition for the current operating environment. Although Hatch expands on our previous work, all the complexity and challenges of abstracting runtime system distribution is novel and only investigated in the context of Hatch.

Hatch makes all distributed design decisions at runtime but requires a human-made local version of the system to explore. In detail, we consider software developers to design and develop standalone local software out of tiny software components. When Hatch starts, it assembles a possible composition of this pre-designed local version of the software by connecting the small software components (this process is named the assembly phase and will later be described in detail). After Hatch assembles a functioning version of the software, it can then explore any distributed software design by relocating or replicating any of the local software constituents small components to other machines which are part of the system's deployment infrastructure, and learn, at runtime, which distributed design composition best suits the operating conditions that the system is being subjected.

Overall, Hatch is a distribution layer, composition builder, and learning agent, shown in Fig. 1. The composition builder and distribution layer find a set of possible compositions of building blocks for a target system, with local and distributed mixtures, and pass these to our learning agent as a list of actions. Hatch also deploys probes into the target system and its environment to observe performance and context.

The distribution layer, in turn, is composed of three sublayers: a transparent and generalized RPC layer, a state management layer and a failure handling layer. The transparent RPC is the part of the distribution layer that enables the actual self-distribution, and because of that this layer is the main focus of this paper. State management and failure han-

dling are key to implement seamless distribution of *any* stateful component, but due to their complexity, their thorough evaluation and detailed description is outside this paper's scope.

Hatch starts by being given an entry-point component for a target system, which will have a 'main' method and a set of required interfaces. Hatch recursively scans the required interfaces and searches for all valid implementations of each one (components); we generally expect multiple such implementations (component variants) to exist such as different sorting algorithms, buffer management protocols, or cache designs. As a result, Hatch finds a set of possible compositions for the target system.

Our distribution layer examines the above list of possible compositions along with a list of available host machines; from this it creates an extended list of compositions that include distributing each possible component of a composition to one of the available hosts. When a component is distributed, the entire sub-tree of the system below that component (i.e., all dependencies, recursively) is also moved and itself represents a set of local compositional choices on the remote host.

We generate a unique identifier for each discovered compositional choice at a given host node (whether that composition is entirely local or includes distributed sub-elements). A list of these unique identifiers is then presented to a reinforcement learning agent, which sees them as a set of *actions* that have (initially unknown) *rewards*. The learning agent operates in an action-observation loop, where it takes an action from this list, waits for a period of time, then collects the reward for that action, before deciding the next best action to take. Whenever a learning agent chooses a composition, Hatch calculates a difference between the current composition and the selected one and performs a sequence of adaptations on the live system to move between the two; if the new composition includes new distribution points, relevant components are loaded at the remote host and a new local learning process is started there.

Hatch supports the distribution of any interface either as a *relocation*, where an interface (and its implementing component and dependency sub-graph) is moved to a remote host, or by *replication*, where an interface is copied across multiple hosts with any associated state replicated or sharded across the copies. All of the machinery for relocation, replication or sharding of a component is created *automatically* by Hatch, with the exception of some types of stateful interfaces/components which need a bespoke state manager plug-in before being replicated or sharded. Because most interfaces and components are designed to be local, Hatch effectively creates a seamless continuity over local/distributed systems, freeing engineers from decision making and analysis of distributed design which Hatch autonomously learns.

Our deployment approach is illustrated in Fig. 1 using the example of a web server, memcached and a database. Each system is discovered as a local entity which is automatically assembled from a set of available building blocks, each of which may have variations available (such as different sorting algorithms or buffer management strategies). These systems are then automatically distributed across remote hosts using our distribution primitives such as relocation or replication.

We now present our generalized approach to seamlessly distribute any component at runtime. We then discuss our distribution primitives, which yield an action/reward matrix for learning, and present details of the learning algorithm.

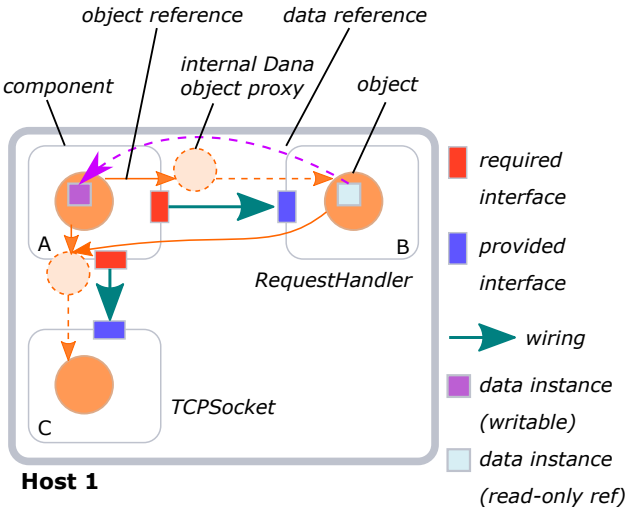
4.1. Generalized distribution of components

Hatch can use two different communication styles when distributing internal system's components. We term these **implicit** and **explicit** communication. At a high level, both are controlled in the same way by Hatch: we take an interface and relocate or replicate its implementing component to a remote host or hosts, whether that interface is a sorting algorithm or an entire database engine (composed of many components). From the abstraction point of view, these two different styles are also seen as simple distribution actions and are not distinguishable by the learning agent. The difference is in how the network communication is achieved, which only has implications on how these components are internally designed.

An **implicit** communication type of component takes a regular Dana required interface, **which was designed to always be local,** and automatically distributes it using our local/distributed continuity approach. This kind of distribution allows Hatch to learn which interfaces are suited to distribution, and to which hosts that distribution is appropriate from the available set, depending on the currently perceived deployment conditions. The cost of doing this is that Hatch needs to provide fully automated fault-tolerance, such that no remote error is permitted to propagate beyond our framework and instead must be immediately recovered from without the system ever being aware that it occurred.

An **explicit** communication type of component is a regular Dana required interface annotated as *distributed*; it is then **explicitly designed with functions that can return remote errors**. Users of this kind of interface would naturally program in such a way that they handle remote error codes, avoiding the need for Hatch to offer automated fault-tolerance for local/distributed continuity. Explicit type of interfaces typically arise when an existing system has already been created (such as a database) and can naturally be wrapped for remote access via a custom on-the-wire protocol, or when distributing implicit communication type of component introduces high performance overhead to the system.

We focus in this section on **implicit** communication proxies, allowing any local interface (and implementing component, along with its sub-graph of dependencies) to be seamlessly distributed. In the rest of this section we examine the core challenges for automated implicit RPC proxies: (i) how to handle references to objects that used to be local but are now remote, including complex reference graphs formed by passing objects as parameters; (ii) what to do with references to data/array instances that used to be local but are now re-




```
interface RequestHandler {
    bool handle(TCPSocket s, HTTPMsg msg)
    }
```

Figure 2: Component and object-level architecture in Dana, showing explicit and implicit internal aspects on a *single host*.

mote; and (iii) how to handle state in distributed components and failures of hosts to which we have distributed code.

4.1.1. Mechanics of self-distribution

We consider the implementation of component building blocks used to describe the mechanics of self-distribution in most object-oriented languages, and we describe such mechanisms using the terminology of object-oriented programming paradigm, which are widely known. However, we highlight the need of a generalized runtime adaptation of running objects for a full implementation of the concepts implemented in Hatch, especially in implementing fully generalized **implicit** communication distribution of any possible component.

We consider here only the most complex scenarios; in our experience the majority of building blocks in a given system require far less supporting infrastructure from Hatch and so are significantly more operationally efficient. To help make the presentation concrete, we use the example in Fig. 2, which shows three components all running in the same host with an object sourced from each one. The provided interface of component B is shown for context, in which we can call functions that pass in references to other objects (such as a connected TCP socket) or references to data instances. We assume that component A instantiates `TCPSocket` objects, for example when accepting client connections, then passes these socket objects into the `handle()` function of `RequestHandler`.

Fig. 2 shows both the user-visible concepts in the language, and the (hidden) sub-structure used by Dana to enable seamless hot-swapping of components [32]. Specifically, Dana uses hidden transparent proxies of objects: whenever an object is instantiated from an interface, the language runtime creates a proxy version of that object which has an internal reference to the real implementation object. References to the proxy can then be passed around to other objects, and if the implementation of the object is hot-swapped this only affects the internal reference between the proxy and its implementation, leaving all other references valid across the hot-swap. The actual hot-swap of behavior is protected by a safe adaptation procedure which guarantees data and control flow integrity in the system so that nothing is lost. Fig. 2 shows a snapshot in time where the interface function `handle()` has been called, with a reference to a `TCPSocket` object passed into it along with a reference to a data instance of type `HTTPMsg`. The figure reflects the fact that proxy objects logically reside at the required interface through which their object was instantiated.

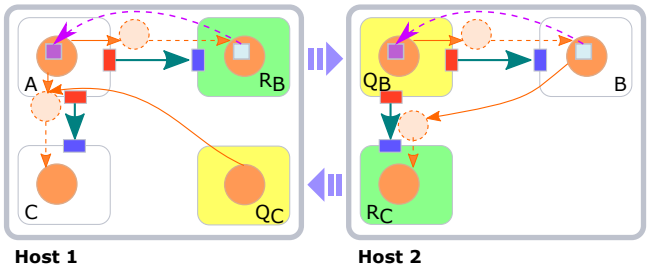


Figure 3: The effect of relocating component B to a different host, which uses two Hatch proxy pairs.

To relocate part of a live system to a remote host, we select a particular required interface R_d of a component, which is currently wired to a local implementation, and replace that local implementation with a Hatch proxy that forwards function calls to a remote location. Object implementations O_S sourced from R_d are *moved* to the remote location along with the implementing component; this is done by instantiating the objects at the remote location and transmitting any state from the original local objects to the remote location. Any references *to* the objects in O_S all still point *at the original proxies* of the objects at the host location they were first created, so that any calls into those objects are then forwarded through the Hatch proxy. Our current proxy implementation uses TCP to carry remote calls and their return values.

The result of relocating component B in Fig. 2 to a different host in shown in Fig. 3. We first create a client (R_B) and server (Q_B) Hatch proxy pair for the `RequestHandler` interface, with the server-side on the remote host. Because some function calls of `RequestHandler` accept instances of `TCPSocket`, we *also* create a Hatch proxy pair for `TCPSocket` in the *reverse* direction (R_C and Q_C). When a `handle()` call is made on `RequestHandler`, the call arrives at Q_B and Hatch automatically instantiates a client proxy for the `TCPSocket` instance passed onto the call; when any `TCPSocket` function is called at B, this call is routed back through R_C to Q_C where it is invoked on the `TCPSocket` object reference at *Host 1*.

For any (read-only) data references held by B after relocation, these data references are held in a lookup table at R_B which watches for any changes made to the fields of each data instance. When a change is detected, the updated data

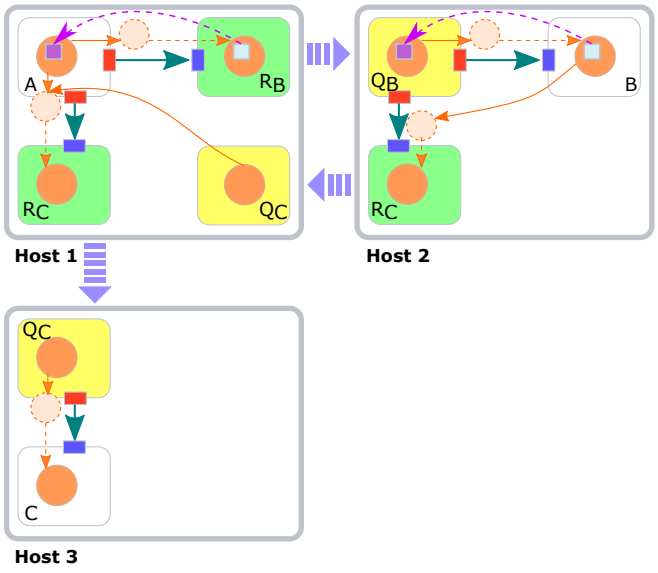


Figure 4: The effect of distributing component *C* to a third host; references to *C* instances still transit via *Host 1*.

instance is sent by R_B to Q_B, which updates the remote copy of the data instance and so changes the values in the data instance as observed by *B*. Data contents can therefore be updated lazily when they occur; because the relative latency of data updates in a local system is itself unpredictable, we consider that this makes no change in the system model.

We next show how the system changes when we further relocate component *C* to a third host, *Host 3*, which is illustrated in Fig 4. Again we introduce a client and server Hatch proxy pair, R_C and Q_C, which forward function calls to the remote host. As above, any references to objects of type *C* held by object *B* at *Host 2* still point at the local object references to *C* on *Host 1*. A function call from *B* to *C* in this case would thus travel from *Host 2* to *Host 1*, then from *Host 1* to *Host 3*. While this seems inefficient, this approach is useful because it allows the implementation of *C* to be changed in a runtime adaptation at a single point of control in the global system, relative to its $A \longrightarrow C$ required interface on *Host 1*; re-wiring this one required interface implicitly causes the new implementation to be used by all references to it. This approach also has a secondary benefit: because network transport pathways follow the component graph (rather than *Host 2* being able to talk directly to *Host 3*) we gain a property of *consistent failure status* as observed from a common perspective. Because all communication to *C* is routed via *Host 1*, it is *Host 1*'s perception of the failure status of *Host 3* that provides a logically consistent picture of failures to the entire system – rather than different hosts having different perceptions of the failure of the same object.

Next we describe the two remaining parts of the distribution layer that enable self-distribution of any stateful component. A large range of system research has addressed failure handling and state management techniques in distributed systems in general [24, 1, 17, 46], and more specifically in RPC contexts [35, 12, 28, 27]. While fully defining state consistency management and failure handling are large research topics in themselves, and are therefore beyond our scope here, we aim to provide a high-level description of how these two important elements of our distribution layer would generally work.

4.1.2. State Management

State management is key to enable self-distribution of stateful components. To fully support seamless relocation, replication and sharding, state management is required to (i) perform state extraction from local running components, and insert state copies into relocated replicas of the original local components; and (ii) keep the remote state copies consistent as the system continues execution. We analyze these points when Hatch performs relocation, replication and sharding; and discuss the key features a programming language needs to provide to enable state management for self-distribution.

Relocation of stateful local components is the simplest action of stateful components' self-distribution, considering this only requires state to be copied and inserted into a remote instance of the component. After component relocation, only one copy of the component and its state exists in the system, thus Hatch is not required to transparently support any consistency models to maintain the system consistent. To enable relocation, a programming language is required to pause incoming function calls to the target component, and enable state to be extracted. In most programming languages, some form of state extraction can be performed through reflection; the Dana in particular supports full state extraction as a standard part of its hot-swappable component architecture.

Replication can be seen as relocation of a single local component to multiple distinct hosts. As such, state extraction can be done just as in the context of relocation. The difference is that state is then inserted into each remote replica as many times as there are replicas. The key challenge in replicating stateful components is then to maintain ongoing consistency across the copies of replicated state. Although this is not the focus of this paper, we envision Hatch transparently adding infrastructure implementing a consistency model between the stateful replicas. Alternatively, to support high availability rather than load-balanced performance, the proxy component can forward all incoming requests to all replicas simultaneously, transparently implementing group communication.

Sharding can be seen as a special type of replication, where the state of the original local component is split in different parts which are individually placed in different replicas. For this, we assume the state is a collection of items such as a list, an array and other similar data types. To enable sharding, Hatch must be able to split the extracted state and place each part of it in a different replica, and for each function called in the local proxy, that proxy must be able to properly find the remote component which hosts the relevant state item (for read operations), or in which replica to (re)place a new item (for write operations). Transparently implementing sharding is particularly challenging, and the

current design of Hatch assumes that plug-ins are created by developers to assist in sharding specific components' state.

4.1.3. Failure Handling

For **implicit** communication type, we assume all component-level behavior is programmed as if it is local, with no programmer-written code to deal with network failure events during function calls. This delegates distributed design choices to real-time learning, but means that network failures must not be 'visible' to the running system, as there will be no specific error handling in place to deal with them.

To support this, Hatch only distributes a component if it can *also* guarantee to be able to silently and automatically recover from any failures that might occur as a result of this distribution, such that we can transparently maintain the impression that a failure never happened. This model implies two different approaches for stateless versus stateful objects.

For stateless objects we simply can detect that a failure occurred and recover the component back to the host with the Hatch client proxy of that component. As an example based on the above section, if component C was distributed to *Host 3*, *Host 1* can detect a failure to communicate with component C on *Host 3* and so can recover from this by re-creating component C on Host 1 and re-firing any function calls that were in-progress against the now-local copy of C.

For stateful objects, each successful function call may change the internal state of an object. Supporting transparent recovery is beyond our scope here, but could be approached using state machine replication via an operation log and occasional full-state backup. Considering the example in Fig. 3, assuming that component B is stateful, we could record all functions called on B in an operation log on *Host 1*, and periodically save the full state of these objects by downloading it from *Host 2* to *Host 1*, resetting the log.

We reiterate the fact that transparent failure handling is outside this paper scope and presents itself as an interesting and important future research avenue. That is mainly because if such transparent fault mechanisms are not provided by Hatch, any autonomous attempt to relocate components across an infrastructure which results in failure will inevitably propagate to the target systems, making it inconsistent and faulty. We believe that such failure handling mechanisms are possible, given that performance may be severely compromised in some cases. We briefly suggested how these mechanisms could be realized in a generalized fashion, and we understand that further research needs to be conducted in order to paint a full picture of the tradeoffs and edge-cases that will impact the system the most.

4.2. Hatch composition building

We build on the above distribution mechanics to support relocation and replication of components, which combine with local behavior alternatives to offer a rich design space for automated design of complex distributed systems.

Relocation uses exactly the mechanics described in Sec. 4.1. This may represent a performance gain if it makes a component closer to a data source, or provides additional compute resource for particularly intensive components.

Replication distribute computation or I/O load across multiple hosts. For stateless components this uses relocation combined with a generic load balancer to either distribute load or shard functions across hosts. To do this, a component C that is currently running locally is simply started on n other hosts, with a load balancer replacing C locally.

For stateful components, Hatch currently requires a case-specific state manager plug-in for the particular component being replicated. Such a state manager may choose for example to lazily propagate state between replicas, or to enforce a single-writer multi-reader approach. We currently assume programmer support is needed to achieve this and Hatch does not replicate stateful components unless it finds a plug-in.

For all distribution styles, Hatch takes the set of original local compositional options discovered from the pool of building blocks, with the set of available hosts to distribute a system over, and generates a set of distributed composition options with relocation and replication, each one represented as a uniquely-identified action for learning.

4.3. Real-time learning

Hatch exposes a complex distributed system design space as a simple set of actions for online reinforcement learning. In general, the goal of online learning is to carefully balance the amount of time spent exploring under-tested actions against exploiting actions already known to perform well [39].

Multi-armed bandits (MAB), originally designed for clinical trials [5, 42], are the best studied theory to balance this tradeoff over time, and are a popular technique for optimization on the web [8, 37]. In our context they are of particular interest because they use very little memory to represent their learning model, require no training before deployment, and use efficient computation for decision-making.

A MAB represents each one of a set of available actions as an *arm*, and assigns a *reward distribution* to each arm. Each time an arm is tried, the expected reward distribution for that arm is updated and the algorithm becomes more confident in this reward distribution. Over time, a MAB will move away from actions with poor reward distribution to focus on those which are best. The exact way in which a MAB models reward distributions, and confidence, varies between algorithms.

In this paper we use bandit algorithm called Upper Confidence Bound (UCB), which models combined confidence over all arms as a logarithmic curve over time. UCB works by taking an action, which in our case causes a live system to adapt to a new composition, including potentially distributing parts of that system to remote host. It then waits a set amount of time, which is a configurable observation window – in our case 5 seconds. After this time, the algorithm collects the reward of the system and updates the average reward for that arm along with the total number of times it has been tried, and the total number of actions that have been taken for the system, from which it derives its relative confidence per arm.

Having updated these values, the algorithm then selects the next arm based on which one has the highest score in a calculation of reward versus confidence as follows:

$$r_k + \sqrt{\left(\frac{2 \ln n}{n_k}\right)}$$

Where n is the total number of actions taken, n_k is the number of times k action has been taken, and r_k is the cumulative average reward for this action. The logarithmic function part of the equation determines the confidence level the learning agent has about a certain action. The higher this value, the smaller level of confidence it has, whereas the lower this value the bigger level of confidence it has. This has the effect that low-confidence arms are tried towards the beginning of learning, while reward levels then tend to dominate action selection as confidence in them grows.

Whenever part of a system is distributed to a remote host by Hatch (either through relocation or replication), Hatch starts a new local learning process on that remote host with its own set of actions relative to the compositional options of the system fragment which has been moved to that host. In our current implementation, these multiple learning processes do not coordinate; instead we assume that decisions taken at high levels of the implicit composition tree tend to be of higher importance while those at lower levels are fine-tuning. We find that this approach enables Hatch to quickly navigate through massive search spaces, by effectively parallelizing the search process across hosts, and in our data center example system only occasionally results in finding a less ideal solution than a globally-coordinated approach would locate.

We also consider special components that are inserted into the system to collect a specific metric from the system as it executes. In our current implementation, Hatch is able to collect the average of response time from any executing composition. The collected metric is used by Hatch as a reward (or cost, in our use case) of the action taken. In our current implementation, the response time metric is used as the cost that the learning algorithm aims to reduce.

5. A self-distributing data center platform

As an example of Hatch in operation we implement a web serving back-end. This is composed of a web server including server-side scripting, memcached implementation, and a database. The web server, memcached analogue, and server-side scripting engine, are all built from scratch, composed from adaptable fine-grained building blocks with a variety of implementation variants – such as different cache replacement algorithms and stream processor pipelines. For the database element we use MySQL, which is wrapped by an **explicit** communication proxy. On top of this infrastructure we have built a Wikipedia-like application on which to test a range of different workloads. In total our system is composed of over 50 potential building blocks; a small subset of this system is illustrated in Fig. 5 showing some of its main features and variation points where alternative blocks are available.

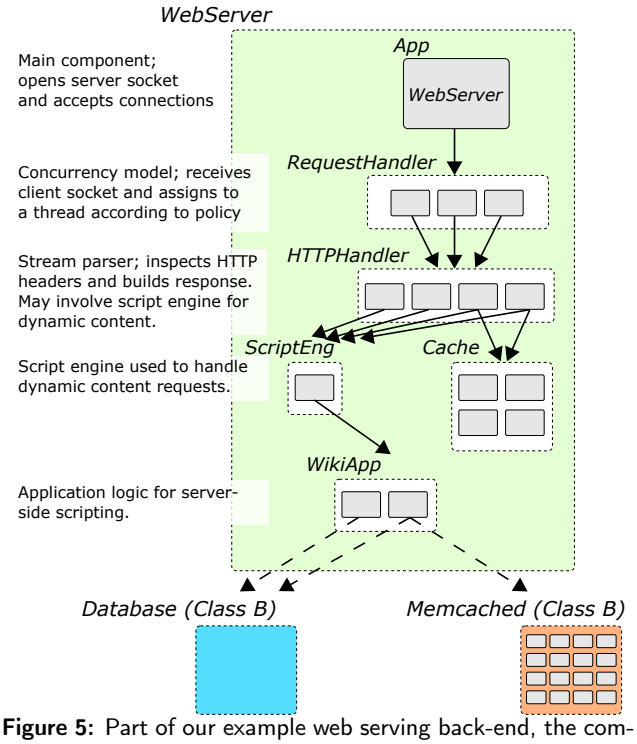


Figure 5: Part of our example web serving back-end, the components of which Hatch autonomously manages to select their local variants and transparently distribute across a set of remote hosts.

Our hosted application running on the web server is similar to Wikipedia, enabling users to create, edit and access articles. Within this application we have two variations of database access sub-systems: one which uses memcached and one which does not. We note that our web serving platform does not explicitly include a load balancer because Hatch can automatically create this effect by replicating and load balancing across any interface(s) in the system.

Hatch is given the main component of the web server system, from which it recursively scans required interfaces to discover all local compositions of building blocks which form a working web server. We transparently inject a measurement probe at the HTTPHandler interface of this system to measure average response time to requests as our relative reward for machine learning. Hatch then constructs a set of distributed compositions for this system, based on the distribution styles discussed above in combination with a set of available hosts.

Whenever an interface is distributed by Hatch to a remote host, that remote host constructs its own set of possible compositions starting from that interface, and takes responsibility for learning the best composition for its delegated part of the system – including further distributing sub-elements of that system area to additional remote hosts.

6. Evaluation

In this section we evaluate our implementation of Hatch, our self-distributing framework on top of a generalized, transparent RPC layer that enables any local system to distribute sub-elements of itself to remote hosts. We evaluate Hatch in two ways: (i) we examine the basic performance of our generalized transparent RPC layer via its implicit automated proxies, which enable our seamless local/distributed continuity; and (ii) we evaluate how Hatch enables us to autonomously learn end-to-end designs of distributed systems. All of our experiments are conducted within a real data center, using a cluster of 10 dedicated rackmount servers which each have Intel Xeon Quad Core 3.60 GHz CPUs and 16 GB of RAM, running Ubuntu Server 18.04.

6.1. Auto-proxy performance

We first analyze the performance of **implicit** communication proxy components in isolation, to understand the relationship between the cost in added latency and the benefit from adding more compute resource. To make distribution effective, network latency added by transparent proxies must be *lower* than compute or I/O latency of the logic being distributed. This relationship also changes under parallelism, where higher levels of concurrent requests benefit from additional CPUs and I/O bandwidth on remote hosts.

We examine this relationship at two specific **implicit** communication type proxies: our HTTP handler interface, and a prime number generator interface. These two points represent the worst, and best, cases for **implicit** proxies in our system.

We first examine the best case, around a component which provides operations with prime numbers (used for example in security handshakes). This component's functions take integers as parameters, and have an exponential increase in processing time for larger integers. To test this, we use a fixed input parameter in function calls that test for primeness, giving it a constant computation time, but vary the level of concurrent requests per second from 10 up to 100. We experiment both with the prime component running locally, and behind a transparent Hatch **implicit** proxy with increasing replication up to 10 servers. The results are show in Fig. 6: as the level of concurrent requests increases, relocating and replicating the component shows increasing performance versus the local case, demonstrating that automated distribution has low enough latency to gain in performance overall.

We now examine the worst case for **implicit** proxies in our data center system, in which this balance can be negative. This is evident in our HTTP handler interface, in which a *handleRequest()* function receives a TCPSocket instance and begins to construct a response, using the *send()* function on the TCPSocket for each part of the response. The request handler potentially interacts with a cache component, file system, server-side scripting component for dynamic content, and compression libraries, depending on the particular composition of behavior in use and the type of request being handled. We specifically examine its performance under

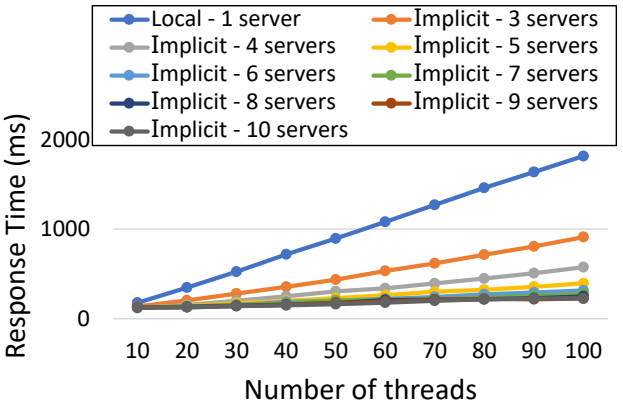


Figure 6: Implicit communication type proxy performance for prime calculations.

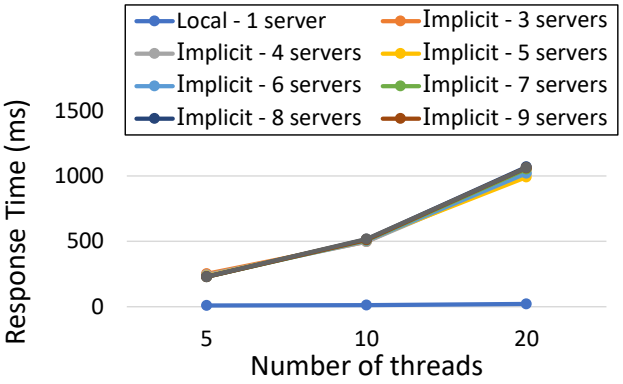


Figure 7: Implicit communication type proxy performance for our HTTPHandler.

purely dynamic content requests at increasing levels of concurrency, using **implicit** proxies to transparently replicate it (and all of its associated dependencies) to an increasing number of hosts. The results of this are shown in Fig. 7.

Here we see that distributing the request handler tends to be worse than having it local. Examining this behavior in detail, we observe that this is because it repeatedly uses the *send()* function on the TCPSocket to incrementally send extra response data from the server-side script system. The local implementation uses the 'direct' version of *send()* against the operating system; in this context *send()* behaves optimistically to the caller in that it returns the number of bytes successfully transferred into the OS send buffer. When *send()* is used over our **implicit** proxy, however, it behaves pessimistically because each invocation of *send()* waits until the remote side acknowledges receipt of the data and returns the local return value (i.e., behaving like an RPC). Because of this conversion from optimistic to pessimistic behavior within *send()*, and the number of times *send()* is called in a server-side script, we therefore see degradation of performance when transparently replicated under **implicit** distribution. This degradation is only avoided if the server-side script is doing very intense processing which dominates the pessimistic *send()* latency.

To overcome this effect, we developed a modified proxy which uses request bursting to collect all response data and send it in a single transmission; this removes the pessimistic *send()* issue and returns the system to yielding positive distribution effects for this component.

6.2. Hatch performance

We now explore the properties of a whole system managed by Hatch, using our entire data center platform against a set of different workloads. Under each workload we first manually execute every composition option of our data center platform (including local composition choices and distribution or replication permutations) to collect ground truth data showing which local and distributed system design options are really best in each set of conditions.

We then run the same workload but using Hatch's real-time reinforcement learning, which has never seen the workload before, to observe the way in which it converges on the best composition as shown by our ground truth tests, and how quickly this convergence happens.

There is a huge array of potential workload combinations available to test which would push Hatch to learn different design choices. We focus here on five major characteristics for a given workload: (i) level of concurrency (simultaneous requests per second); (ii) average processing time of a dynamic content request; (iii) number of dynamic content requests per second; (iv) number of static content requests per second; and (v) content types of static content requests. Our evaluation examines the effects of different ratios of these characteristics.

To aid in making clear inferences between cause and effect in each experiment, we also restrict the points in our target system for which Hatch proxies can be inserted. We focus specifically on components that implement different ways to process HTTP requests, enabling relocation and replication of such components; besides the Hatch distribution options, these components also have a range of local implementations such as caching static content, compressing content, caching and compressing static content, or always accessing disk.

We begin with experiments exploring synthetic ratios of different workload characteristics to demonstrate their effects, and finish with realistic mixtures based on a real-world trace.

6.2.1. Ground truth

Our first experiments examine predominantly dynamic content requests, which are handled by the web server's server-side scripting component, under increasing concurrency. The results are shown in Fig. 8, which is split into two phases: in the first the system is subjected to 5 dynamic content requests per second, and in the second phase to 20. For both phases we show an all-local composition of our web serving system running on a single host, and two different distribution compositions which are indicative of the range of performance characteristics available through different distributed system designs. In the first phase of this experiment, keeping everything in the web serving system on a single host (i.e., local)

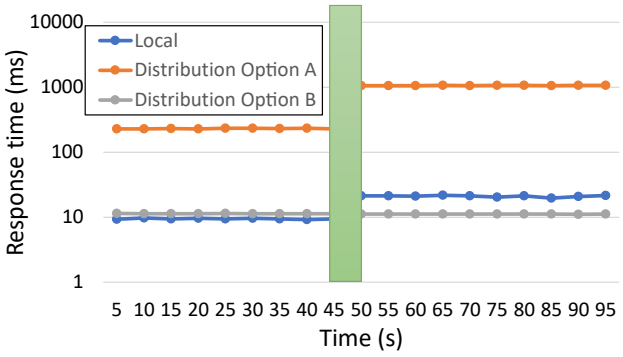


Figure 8: Ground truth for dynamic content workloads

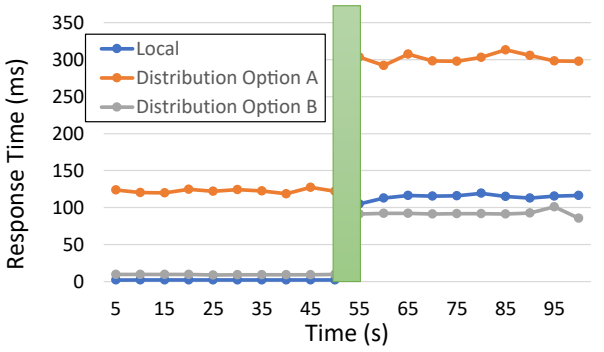


Figure 9: Ground truth for realistic static/dynamic mixtures.

is the best option. This is because the single server has sufficient cores to deal with this level of concurrency without negative side effects. In the second phase, this level of concurrency exceeds the core count of the single server and so computation becomes a dominant cost above the latency between two hosts in our data center. In this case it then becomes beneficial to replicate the request-handling part of the server to additional machines. The exact point at which this transition becomes useful depends on the type of processing in server-side scripting, and specific hardware characteristics – a key motivation for a learned solution.

Next we examine the effect of different ratios of static and dynamic content. This has a key interaction with how distributed our system becomes. We observe that if *all* content was static then hosting all parts of our platform on a single server would be best. This is because Hatch effectively converts a system to a layer-7 load balancer in this case, and the added latency of replication is more than reading static content from a local disk. We therefore examine the ratio of dynamic/static content which provokes distribution to be best.

Fig. 9 shows the result of the system in different compositions when subjected to two workloads. The first uses the ratio of static to dynamic content of a public Wikipedia trace [43], in which the vast majority of requests are reads and have a ratio of 54% images, 31% text content, and 17% dynamic content. Here we see that the best composition is actually to have everything local, since enough content is static.

The second workload, on the right of in Fig. 9, shows the same trace in which we increase the processing time of each dynamic request, reflecting a slightly higher CPU intensity per page or a higher proportion of write requests. Here we see a reversal of the above result, where replicating the request handler to multiple servers becomes the ideal solution.

6.2.2. Learning

We now examine how real-time reinforcement learning can, starting from no initial information, learn the ideal composition and distribution of a system. In all of these experiments our systems start on a single entry-point machine to our data center, and are then subjected to various different workloads. In response to these workloads, Hatch begins to explore the design space in real-time, seamlessly adapting between the available local and distributed composition options, and spreading Hatch learning agents themselves across multiple hosts. For each of these experiments Hatch has a total search space of 699, 044 unique possible ways to compose the system, which are seamlessly explored in real-time.

We first use a workload consisting only of dynamic content at a constant rate of at least 20 simultaneous requests and above, matching one of our ground truth data points. Fig. 10 and Fig.11 show the results of this under learning, along with the data from our ground truth tests. Both graphs show average response time to client requests in milliseconds on the y-axis, and time on the x-axis, and the response time of our data center system under Hatch learning control together with an all-local composition and two selected distributed compositions from our ground truth experiments. Fig. 10 shows our initial learning system at work, and highlights a common effect of learning over reward distributions: although Hatch quickly eliminates the worst-performing options, it is more difficult to discriminate between more similar-looking actions in terms of the overall reward landscape. This is mainly caused by the large difference between two particular distributed composition options, shown on the graph as Option A and Option B; the very large difference in reward here causes normalization to skew the relative impact of this difference so that the smaller differences between higher-performing compositions are more difficult for the learning algorithm to focus on. To correct this we use adaptive normalization which avoids poor actions that have high confidence, yielding the results shown in Fig.11, in which Hatch quickly learns to get close to the ideal solution in about 5 minutes (note that Hatch does select the same composition as the ground truth here, though its response time appears slightly higher due to noise).

Finally we examine performance under our Wikipedia-like workloads, which have a more natural mixture of static and dynamic content requests, with the results shown in Fig. 12, again showing two selected ground truth compositions for reference along with the all-local composition of the data center system and our learning trace. Here we see that Hatch converges quickly on a good solution, using a distributed replication design for the HTTP handler component, and actually found a better composition of this design during learning than we found in our ground truth experiments. This is because the very high number of possible compositions is too large for us to test during ground truth experiments, and so we are only able to try a sub-set. The result of this experiment therefore shows that Hatch is able to quickly navigate this search space to find a better specific distributed system design than our ground truth experiments located.

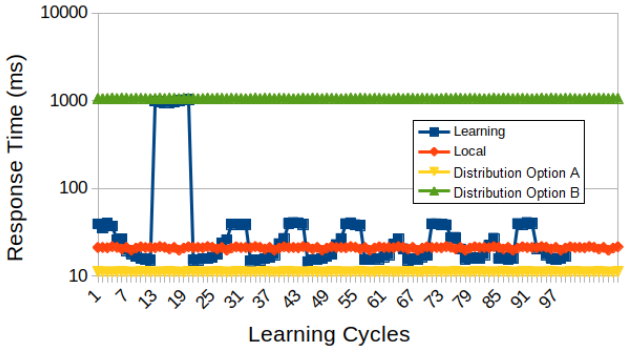


Figure 10: Hatch learning with dynamic content workloads.

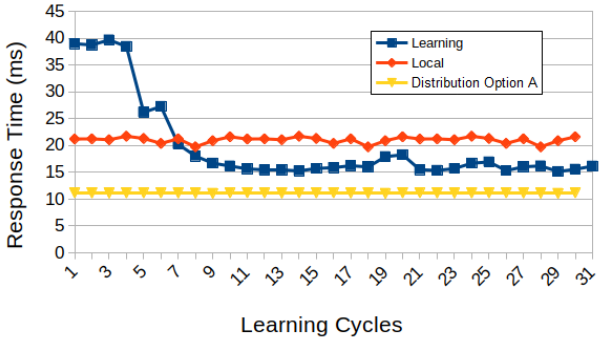


Figure 11: Hatch learning with adaptive normalization.

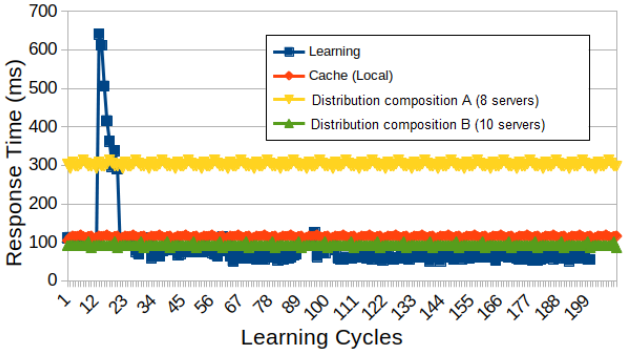


Figure 12: Hatch learning with a Wikipedia-like workload.

7. Discussion

In this section we discuss some of the limitations and assumptions of our approach, and areas for future work. We first examine the characteristics of our application domain in this paper and reflect on which kinds of application are likely to be suited to Hatch. We then examine a broader set of challenges that our work in this paper does not address, and

discuss how those challenges may be tackled in the future.

In this paper we evaluated Hatch in the context of data center web-based services, in which a system processes a stream of requests from users and the overall performance of the system can be reasonably measured in terms of how quickly requests are serviced. The systems that we examine were also specifically designed only as local systems operating on a single host, and therefore benefit from the automated distribution offered by Hatch.

The use of a stream-processing system enables Hatch to quickly collect metrics about the system in short periods of time (taking measurements every few seconds), and to use these metrics to optimize the system. The use of Hatch in batch processing systems introduces potential challenges in the measurement of ongoing performance in order to make rapid progress in finding an ideal solution; in some cases incremental progress through a batch job can serve this purpose (e.g. [10], but identifying effective sub-batch measurement approaches can be challenging. On the specific optimization metric used in this work, to date we have only experimented with response time as a metric and have only examined a single metric. The use of other measurements besides response time may present their own challenges, such as latency between action and effect during machine learning, and the use of multiple metrics is a topic of increasing study in the wider autonomous systems community. Despite these limitations, our use of a single metric with a stream-processing system represents a broad class of modern systems – the majority of web-systems today fit this criteria, but other systems such as code running on mobile phones or desktop computers could also benefit from the ability to selectively offloaded code to remote sites for enhanced responsiveness. In future work we intend to further examine other domains to which Hatch may be applied to learn broader lessons on these issues.

On more specific limitations, we next examine issues around failure handling, state management, machine learning, multiple-instances, and system evolution.

Failure recovering mechanisms are an important part of Hatch. Although the detailed definition and evaluation of such mechanisms is beyond the scope of this paper, we identify the general concept of how they could be implemented via transparent recovery. Even if such a general mechanism is implemented, we note that there is a trade-off here in how much overhead they add relative to the performance gain of distributed code to a remote location: the specific limits of this trade-off are an interesting topic for future work.

On state management for replication or sharding, again in some contexts the necessary maintenance processes for state consistency may add prohibited overhead to the system performance. As well as further examining which kinds of component are likely to benefit from automated replication, versus those that may present too high an overhead instate maintenance, it is also worth considering that some systems may inherently have different levels of tolerance for data staleness and inconsistency. For these kind of systems, a more flexible state consistency model could be made explicit to allow Hatch to relax state consistency where possible and gain further performance improvements.

On machine learning, in this paper we demonstrate search spaces for real-time navigation that are on the order of hundreds of thousands of permutations. The speed with which this search space is navigated is a key challenge, and higher-order search spaces as a system scales up would further exacerbate this. As Hatch takes longer to decide the most suitable design choice for the system, the longer Hatch experiments with sub-optimal choices that affect the systems performance, thus becoming an important issue to address. Fortunately, in the literature there are some promising approaches to tackle this issue. Ontanón [29] is a notable example and proposes a method that divides the action space and explores it smartly and effectively, demonstrating their approach to scale to millions of actions. Another promising approach is the combination of deep neural networks to reduce the search space of actions as Donckt et al. reports in their work [44]. It is also possible that the individual contributions of each component to the system can be isolated and transferred to estimates for untried compositions, in a similar way to how genetic algorithms attempt to isolate which genes contribute positively or negatively to a solution; this may allow a large number of possible actions to be safely ignored based on some similarity to actions that have already been tried.

On the potential for multiple Hatch instances operating on the same pool of resources, in this paper we have assumed that only a single Hatch instance is active over a given resource pool. In practice it is likely that multiple Hatch instances would operate in each data centre and may have access to the same resources. Besides obvious approaches such as resource containers, which limit how much resource can be used by each process resident on a host, it may also be of interest to examine how multiple Hatch instances could communicate in order to mutually move towards an equitable or ideal usage of available shared resource.

Finally, on the issue of system evolution, in this paper we have assumed that all components for use in the system are available when that system first starts. While this is a useful simplifying assumption, we would also expect new component variants to be able to arrive later (as engineering teams have new ideas based on how the system is behaving) and for Hatch to be able to learn whether these new arrivals are useful. The main challenge here is in understanding the statistical comparisons in machine learning between actions that have existed for a long time (and about which we have high confidence in their respective rewards), versus newly-available actions that are untested. Many multi-armed bandit implementations would naturally focus on newly-available actions heavily to understand how they behave, until equal confidence is available in that behaviour, but further research on this specific area would help to understand the full set of implications of software evolution combined with real-time learning.

8. Conclusion

We have proposed the concept of self-distributing systems, in which any local system can have its constituent parts automatically and transparently distributed or replicated through a network. This helps to automate distributed system design by learning what to distribute, how, and to where.

We have implemented this concept in Hatch, along with a complete end-to-end data center back-end system as an example of Hatch's capabilities. Hatch discovers a set of possible ways to construct a target system, then injects potential distributed designs into those possibilities using its transparent distribution layer; all of this potential is represented as a simple set of actions for real-time reinforcement learning, which explores the design space of a live system in real-time to optimize against its actual deployment conditions.

To the best of our knowledge, Hatch is the first approach to fully generalized and automated design of a distributed system. Our evaluation demonstrates that there is a diverse set of different optimal points in a given design space for our web serving platform depending on workload, and that Hatch can rapidly find those design points in a live production setting.

In future work, we will apply Hatch to a range of system types beyond data center back-ends to gain a deeper understanding of the value of autonomous local/distributed continuity, and will also examine how real-time reinforcement learning approaches can scale up to very large decision spaces. Vast action space exploration is central to our approach and still remains an open issue. Finally, we will investigate in detail state management and failure handling in implicit communication type of components and evaluate these two dimensions in different scenarios.

Acknowledgments

This work was supported by the UK Leverhulme Trust via the Self-Aware Datacentre project, grant RPG-2017-166. Dr Rodrigues Filho is also funded by São Paulo Research Foundation under the grant 2020/07193-2. This re-search is also part of the INCT of the Future Internet for Smart Cities funded by CNPq proc.465446/2014-0, CAPES proc.88887. 136422/2017- 00, and FAPESP procs.14/50937-1 and 15/24 485-9.

References

[1] Aksoy, R.C., Kapritsos, M., 2019. Aegean: Replication beyond the client-server model, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, USA. p. 385–398. URL: https://doi.org/10.1145/3341301.3359663, doi:10.1145/3341301.3359663.

[2] Alipourfard, O., Liu, H.H., Chen, J., Venkataraman, S., Yu, M., Zhang, M., 2017. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA. pp. 469–482.

[3] Ardelean, D., Diwan, A., Erdman, C., 2018. Performance analysis of cloud applications, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), USENIX Association, Renton, WA. pp. 405–417.

[4] Bennani, T., Blain, L., Courtes, L., Fabre, J.C., Killijian, M.O., Marsden, E., Taiani, F., 2004. Implementing simple replication protocols using corba portable interceptors and java serialization, in: Proceedings of the 2004 International Conference on Dependable Systems and Networks, IEEE Computer Society, Washington, DC, USA.

[5] Berry, D.A., Fristedt, B., 1985. Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability). Springer.

[6] Binnig, C., Crotty, A., Galakatos, A., Kraska, T., Zamanian, E., 2016. The end of slow networks: It's time for a redesign. Proc. VLDB Endow. 9, 528–539. doi:10.14778/2904483.2904485.

[7] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B., 2004. An open component model and its support in java, in: Component-Based Software Engineering, Springer Berlin Heidelberg. pp. 7–22. doi:10.1007/978-3-540-24774-6_3.

[8] Chapelle, O., Li, L., 2011. An empirical evaluation of thompson sampling, in: Advances in neural information processing systems, pp. 2249–2257.

[9] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T., 2008. A generic component model for building systems software. ACM Trans. on Comp. Systems 26, 1:1–1:42.

[10] Dean, P., Porter, B., 2021. The design space of emergent scheduling for distributed execution frameworks, in: 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 186–195. doi:10.1109/SEAMS51251.2021.00032.

[11] Diao, Y., Hellerstein, J.L., Parekh, S., Bigus, J.P., 2003. Managing web server performance with autotune agents. IBM Syst. J. 42, 136–149. doi:10.1147/SJ.2003.5386833.

[12] Djilali, S., Herault, T., Lodygensky, O., Morlier, T., Fedak, G., Cappello, F., 2004. Rpc-v: Toward fault-tolerant rpc for internet connected desktop grids with volatile nodes, in: SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, pp. 39–39.

[13] Dragojević, A., Narayanan, D., Castro, M., Hodson, O., 2014. Farm: Fast remote memory, in: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), USENIX Association, Seattle, WA. pp. 401–414.

[14] Dragojević, A., Narayanan, D., Nightingale, E.B., Renzelmann, M., Shamis, A., Badam, A., Castro, M., 2015. No compromises: Distributed transactions with consistency, availability, and performance, in: Proceedings of the 25th Symposium on Operating Systems Principles, ACM, New York, NY, USA. pp. 54–70. doi:10.1145/2815400.2815425.

[15] Gao, J., 2014. Machine learning applications for data center optimization. Google White Paper .

[16] Gao, P.X., Narayan, A., Karandikar, S., Carreira, J., Han, S., Agarwal, R., Ratnasamy, S., Shenker, S., 2016. Network requirements for resource disaggregation, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA. pp. 249–264.

[17] Guerraoui, R., Pavlovic, M., Seredinschi, D.A., 2016. Incremental consistency guarantees for replicated objects, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 169–184.

[18] Haase, J., Eschmann, F., Klauer, B., Waldschmidt, K., 2004. The sdvm: A self distributing virtual machine for computer clusters, in: International Conference on Architecture of Computing Systems, Springer. pp. 9–19.

[19] Haase, J., Hofmann, A., Waldschmidt, K., 2010. A self distributing virtual machine for adaptive multicore environments. International journal of parallel programming 38, 19–37.

[20] Hoffmann, M., Lattuada, A., Liagouris, J., Kalavri, V., Dimitrova, D., Wicki, S., Chothia, Z., Roscoe, T., 2018. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows, in: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), USENIX Association, Renton, WA. pp. 95–110.

[21] Jiang, J., Sun, S., Sekar, V., Zhang, H., 2017. Pytheas: Enabling data-driven quality of experience optimization using group-

based exploration-exploitation, in: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), USENIX Association, Boston, MA. pp. 393–406.

[22] Jindal, A., Hu, Y.C., 2018. Differential energy profiling: Energy optimization via diffing similar apps, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA.

[23] Kalia, A., Kaminsky, M., Andersen, D., 2019. Datacenter RPCs can be general and fast, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA. pp. 1–16.

[24] Liu, S., Viotti, P., Cachin, C., Quéma, V., Vukolić, M., 2016. {XFT}: Practical fault tolerance beyond crashes, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 485–500.

[25] Maassen, J., van Nieuwpoort, R., Veldema, R., Bal, H.E., Plaat, A., 1999. An efficient implementation of java's remote method invocation. SIGPLAN Not. 34, 173–182. doi:10.1145/329366.301120.

[26] Moore, R., Klauer, B., Waldschmidt, K., 2000. Tailoring a self-distributing architecture to a cluster computer environment, in: Proceedings 8th Euromicro Workshop on Parallel and Distributed Processing, IEEE. pp. 150–157.

[27] Narasimhan, N., Moser, L.E., 2001. Transparent Fault Tolerance for Java Remote Method Invocation. Ph.D. thesis. AAI3016401.

[28] Narasimhan, P., Moser, L.E., Melliar-Smith, P.M., 2002. Strongly consistent replication and recovery of fault-tolerant corba applications. Comput. Syst. Sci. Eng. 17, 103–114.

[29] Ontanón, S., 2017. Combinatorial multi-armed bandits for real-time strategy games. Journal of Artificial Intelligence Research 58, 665–702.

[30] OSGI, Alliance:. https://www.osgi.org/.

[31] Peng, Y., Bao, Y., Chen, Y., Wu, C., Guo, C., 2018. Optimus: An efficient dynamic resource scheduler for deep learning clusters, in: Proceedings of the Thirteenth EuroSys Conference, Association for Computing Machinery, New York, NY, USA.

[32] Porter, B., 2014. Runtime modularity in complex structures: A component model for fine grained runtime adaptation, in: Component-Based Software Engineering, ACM. pp. 26–32.

[33] Porter, B., Grieves, M., Rodrigues Filho, R., Leslie, D., 2016. RE^{X}: A development platform and online learning approach for runtime emergent software systems, in: Symposium on Operating Systems Design and Implementation, USENIX. pp. 333–348.

[34] Porter, B., Rodrigues Filho, R., 2021. A programming language for sound self-adaptive systems, in: 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS).

[35] Reis, D., Miranda, H., 2012. Transparently increasing rmi fault tolerance. SIGAPP Appl. Comput. Rev. 12, 18–26. URL: https://doi.org/10.1145/2340416.2340418, doi:10.1145/2340416.2340418.

[36] Rodrigues Filho, R., Porter, B., 2017. Defining emergent software using continuous self-assembly, perception, and learning. Transactions on Autonomous and Adaptive Systems 12, 1–25.

[37] Scott, S.L., 2010. A modern bayesian look at the multi-armed bandit. Applied Stochastic Models in Business and Industry 26, 639–658.

[38] Shan, Y., Huang, Y., Chen, Y., Zhang, Y., 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association, Carlsbad, CA. pp. 69–87.

[39] Sutton, R.S., Barto, A.G., 2018. Reinforcement Learning. 2nd ed., MIT Press, Cambridge, MA, USA.

[40] Tammana, P., Agarwal, R., Lee, M., 2016. Simplifying datacenter network debugging with pathdump, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA. pp. 233–248.

[41] Tejedor, E., Badia, R.M., 2008. Comp superscalar: Bringing grid superscalar and gcm together, in: 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), pp. 185–193. doi:10.1109/CCGRID.2008.104.

[42] Thompson, W.R., 1933. On the Likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika 25, 285–294. doi:10.1093/biomet/25.3-4.285.

[43] Urdaneta, G., Pierre, G., Van Steen, M., 2009. Wikipedia workload analysis for decentralized hosting. Computer Networks 53, 1830–1845.

[44] Van Der Donckt, J., Weyns, D., Quin, F., Van Der Donckt, J., Michiels, S., 2020. Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals, in: Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 20–30.

[45] Veeraraghavan, K., Meza, J., Chou, D., Kim, W., Margulis, S., Michelson, S., Nishtala, R., Obenshain, D., Perelman, D., Song, Y.J., 2016. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services., in: OSDI, USENIX. pp. 635–651.

[46] Vogels, W., 2009. Eventually consistent. Communications of the ACM 52, 40–44.

[47] Waldo, J., Wyant, G., Wollrath, A., Kendall, S., 1994. A Note on Distributed Computing. Technical Report. Mountain View, CA, USA.

[48] Woodfin, T.R., 2002. Self-distributing computation. Ph.D. thesis. Massachusetts Institute of Technology.

[49] Zhang, Y., Huang, Y., 2019. "learned": Operating systems. SIGOPS Operating Systems Review 53, 40–45. URL: https://doi.org/10.1145/3352020.3352027, doi:10.1145/3352020.3352027.

[50] Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., Stumm, M., 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle, in: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA. pp. 603–618.