Adaptable Actors: Just What The World Needs

Paul Harvey
National Institue of Informatics
Tokyo, Japan
paul@nii.ac.jp

Joseph Sventek
University of Oregon
Eugene, Oregon, USA
jsventek@uoregon.edu

Abstract

The combination of improved battery technology and more power-efficient computing hardware has led to the proliferation of heterogeneous distributed systems. This *internet of things* consists of embedded, wearable, hobbyist, parallel, and commodity devices. Given the different resource and power constraints of such systems, applications must be able to reconfigure or adapt their runtime execution environment in order to make best use of the resources available. In order for the underlying operating system/runtime to support runtime adaptation, an application must be suitably designed. The actor model of computation presents a natural fit for programming such adaptive systems with shared-nothing semantics and use of message passing. This paper addresses the limitations of current actor approaches and argues that an actor is the appropriate *unit of adaptation*. This is justified through experimentation across heterogeneous platforms.

CCS Concepts • **Software and its engineering → Operating systems**; **General programming languages**;

Keywords adaptation, actors, heterogeneous hardware

1 Introduction

There are now many different computing platforms. From machines consisting of multicore CPUs and co-processors, such as GPUs, to small, embedded, battery-powered computers. Mobile phones and tablet computers have become computing platforms in their own right. Given the diverse range of resource-constraints and operating conditions associated with this *internet of things*, applications can no longer be static, and must be able to adapt to their execution environment to make best use of the resources at hand. As these runtime environments evolve, applications must also be able to discover resources, as static configuration information is no longer suitable. The motivation to adapt applications at runtime is seen in many different areas: wireless sensor networks (WSNs) which are deployed in dangerous areas, efficient use of machines in data centres as system load varies, or users who expect desktop performance from tablet machines without the power and performance cost. Adaptation is challenging as different equivalence classes of hardware devices each come with their own programming styles and runtimes, meaning that developers must have working knowledge of a number of different programming styles, hardware platforms, and networking technologies, which is already proving an issue [19].

There are many approaches to this problem. Some require entire applications to be modified in a single step, whereas most involve loosely-coupled software components which explicitly communicate. These approaches vary in the types of platforms they support, and provide little or no support for runtime discovery of resources. Furthermore, one of the main limitations of loosely-coupled systems is that the *user* must ensure that an application is suitably decoupled. This is error-prone, and can be a barrier to adoption by non-experts.

Alternatively, in the actor-model of computation, applications are composed of shared-nothing loci of computation which interact via explicit message passing by design. This work extends an actor-based approach to address the challenges of adaptation and communication across heterogeneous hardware platforms.

This paper makes the following contributions: 1) enabling on-demand discovery of software or hardware resources using language-based queries, 2) enabling runtime adaptation of actors, and 3) evaluation of these extensions on a range of hardware platforms for both exemplar and realistic applications. The results show that an actor-based model enables straightforward programming of adaptive applications without any sizeable performance cost, and that the actor is the appropriate *unit of adaptation*. Note that the language and runtime used are for convenience and that this hypothesis is independent of the specific implementation.

This paper is organised as follows: Section 2 overviews the state of the art, 3 describes the actor framework, 4 details the design and implementation of the extensions, 5 describes the experimental results, and 6 summarises the work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLOS'17, October 28, 2017, Shanghai, China

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5153-9/17/10...$15.00
https://doi.org/10.1145/3144555.3144559

2 Related Work

Much work was done on process migration and service discovery [17] in the early 20th century, however, hardware platforms have evolved since that time. Due to limited space, the following discusses the most recent and relevant contributions with regards to runtime adaptation for context.

WSNs use two adaptation approaches: image-based and module-based. Image-based approaches replace both an application and the majority of the runtime [14]. This coarse-grained approach incurs a large power cost and requires that the hardware is reset, losing any data in RAM. Module-based approaches [13] replace individual application elements, but do not support migration and/or require homogeneous hardware platforms. At the opposite end of the scale, Xen [7] is used in data centres to facilitate adaptation in terms of OS instances, although this relies on a closed networking environment, a distributed file system, and homogeneous hardware. Mirage [16] is based on Xen, and offers this service at the application level, but still suffers from these constraints.

Giurgiu et al. [9] extended a system for mobile cloud computing. It is Java-based and uses OSGi [18] bundles to enable remote computation in the cloud. As with other component-based approaches, this work requires *users* to create loosely-coupled code. The work doesn't support on-demand discovery of resources, and relies on a static cloud infrastructure. MPI [8] supports remote process creation and location-transparent communication, although users must manually marshal/demarshal complex data types. It supports static *hardware* discovery based on predefined configuration files, but isn't designed for a dynamic execution environment with transient nodes, and doesn't support process migration. Emerald [15] was developed in the 1980's, and was one of the first systems to explore language-integrated runtime adaptation. It was designed for homogeneous desktop platforms. In Emerald everything is an object, and all objects could be adapted to other locations. While it supports object discovery, this requires knowledge of the exact object to be found, offering limited flexibility. Emerald has influenced this work.

Regarding actor-based approaches, Erlang [2] is a popular actor-based system, targeted at high-throughput streaming applications. While it supports location transparent communication, and remote actor creation, it does not support migration or runtime discovery of resources, and is not suitable for embedded platforms. A similar set of arguments can be made for Scala [10]. Although it does support runtime discovery, developers must specify Unix-style path names to gain references to a specific actor. This work presents a much more flexible mechanism. Salsa [20] is an actor-based system which supports actor migration. However, this requires the pre-deployment of class files at each site to which actors may be migrated, limiting the flexibility of this approach. Salsa is not supported on embedded hardware. CAF [6] is an actor-based library, which uses template metaprogramming to integrate with C++. CAF does not support runtime discovery, or **strong** migration [3] of actors. Finally, Orleans [4] is a novel actor-based implementation, designed for a data centre environment, requiring a highly provisioned fixed infrastructure, unlike this work.

3 Ensemble

Ensemble [11] is an imperative actor-based programming language. It was originally created to ease the development of software for WSNs. An Ensemble actor has its own private state and a single thread of control. The thread of control is expressed as a `behaviour` clause. The code within this clause is repeated until explicitly told to stop. All actors execute within a `stage`. A stage represents a memory space, thus many stages may exist per physical machine. Ensemble supports reference counted garbage collection and exceptions.

Actors have *shared-nothing* semantics - i.e., they share no state. Message passing along uni-directional, typed channels is used for actor information sharing. Each channel may have an optional buffer to enable asynchrony during communication. When no buffer is specified or a buffer is full, the system reverts to synchronous, blocking communication. Channels are either statically associated with a specific actor via an interface, or dynamically created and composed at runtime into one or more *1-1*, *1-n*, or *n-1* topologies.

Ensemble applications are compiled to Java source code, and then custom Java class files for use with the custom Ensemble VM [5]. Each VM instance can be thought of as a stage. Ensemble applications can be executed on desktop, parallel accelerators (e.g., GPUs), Raspberry Pi (RPi), Lego NXT, and Tmote Sky hardware platforms, and use a range of networking technologies.

4 Adaptation Through Actors

The shared-nothing semantics and explicit message passing of the actor model provide the perfect structure to facilitate runtime adaptation. Consequently, the Ensemble language and runtime was modified to enable runtime discovery of actors and stages, location transparent communication via channels, as well as remote spawn and strong migration of actors across Ensemble supported platforms.

4.1 Runtime Discovery

Unlike systems which gain references to resources based on network addresses, or names, it was desirable to enable developers to specify the **properties** of sought resources. By having the runtime provide such a mechanism, it gives the developer the flexibility that a network address does not. This work enables the discovery of stages and actors.

Properties Within the language, a new `property` type was defined. It is a struct which represents a key-value pair, where the key is a string, and the value is a generic, primitive type. Once an actor has either created or been sent an array

of properties, it may `publish` them. Publishing associates the set of properties with the actor to be used in discovery, and makes the actor visible to the discovery process; by default, an actor is invisible. An actor may `unpublish` to become invisible. An actor's properties can be changed by reissuing a `publish` with a new set of properties. To ensure state encapsulation, an actor may only perform these operations on itself. Attempting to publish an array containing multiple properties with the same name generates an exception. As this work uses an on-demand approach, a `publish` records the list of properties and the visibility of the actor at the **local** stage, discussed below.

Unlike actors, stages are always visible, and have a static set of predefined properties, as listed below. Apart from **NAME**, when queried, the current value for the specified key will be returned, rather than a predefined value. These are intended as a first attempt at a useful set of properties. This information enables developers to programmatically make informed decisions about runtime adaptation.

- **NAME**: Name of this stage - string.
- **#CORES**: Number of CPU cores available - integer.
- **#AVAIL_RAM**: Free RAM in bytes - integer.
- **#ACTORS**: Current number of resident actors - integer.
- **#OPENCL**: Are OpenCL actors supported - boolean.
- **#DISTANCE**: Physical distance between querying and queried stage, uses bluetooth RSSI - integer.
- **#CPU_LOAD**: Current % load across all CPUs - integer.

Query To locate a stage or actor, a developer must define a query (line 4)[1] . Although there are languages (e.g. SQL) which are designed to specify queries, an Ensemble-specific format was chosen to reduce the amount of effort required by the developer.

A `query` is an addition to the language. It is a named, first-class entity, which may be sent along channels. A query is similar to a function; it may have arguments, but only returns a query type. The query body consists of a boolean expression.The expression may only consist of primitive values to match the constraint on the `property` values. There are two exceptions: *remote keys*, and `can_execute()`.

Remote keys are prefixed by a $, and are placeholders for the values within the properties of the actors that the query will be applied to. During discovery, these keys are replaced by values if the key is found in a property, or the relevant clause yields `false`, thus permitting reasoning about unmatched keys. The `can_execute()` operator is provided to determine if a remote stage can execute the specified actor - e.g., does the stage have sufficient RAM for the necessary class files? A query is represented at runtime as a compiler-generated bytecode program, and is executed at the (remote or local) stage being queried.

Discovery Discovery of actors enables channel connections, so that actors can communicate. Discovery of stages enables runtime adaptation of actors. Once a query has been defined, published actors must be explicitly discovered via the `findActors()` procedure. This procedure takes an `interface` type and a `query` type. The interface ensures that the channels of a discovered actor can be safely and correctly accessed, and also filters ineligible actors before the query is applied. An actor is defined as supporting the specified interface if it *structurally* matches, as opposed to name-based equivalence. Structural matching was chosen as applications are expected to be compiled independently and still interoperate, hence, the functional correctness and type safety of structural equivalence was chosen. Semantic correctness can be achieved by the user-defined `query`.

`findActors()` transmits the specified `query` and `interface` to all in-range devices via available networking technologies. This multicast approach was chosen for two reasons. Firstly, given the different equivalence classes of hardware devices, the different networking hardware that they use, and the fact that some hardware platforms may be physically mobile, it was not practicable to assume that there would be a central *oracle* (name server/broker) as in other approaches [1, 8]. Secondly, considering adaptation, a centralised repository of data may become outdated quickly, or would require a large amount of network communication to keep it updated. Note that the actor model presented does not dictate this approach, and it would be equivalent to use a dedicated infrastructure.

Once a discovery request is received by a stage, the specified interface is compared against all published actors. Only matching actors may then be *queried*. An actor may have multiple interfaces, but only one need match. Next, the boolean expression of the query is evaluated against the properties of each eligible actor. If the name and type of a remote key and the key of an actor's property match, the corresponding value is used in place of the remote key in the query. If there is no match, or the types do not match, that clause of the expression evaluates to false. This is useful as different actors may have similarly named properties with different types. If the query evaluates to true, a remote reference is constructed for the actor and any of the channels of its matched interface.

Once all appropriate references have been constructed, they are sent back to the initial querying actor. At the initiating stage, an array is created and populated with any received references. This array may be empty if no replies are received before the (configurable) discovery timeout fires. This array represents the return value of the search. The VM will also query the **local** published actors/stage using the same process. Any eligible actors are added to the array.

In the language, an array of `interfaces` of the type passed to `findActors()` is returned to the calling actor. As in the existing system, each interface is a reference to a unique actor, although now, each reference may be local or remote. As the type and number of available channels in the interface specified in the discovery are known at *compiletime*,

[1] This and the following line numbers refer to Listing 1, Section 5.2

the compiler will generate errors if the discovered actor references or the channels dereferenced from them are used improperly. Discovering stages is a similar process, except that no interface is specified (line 10).

4.2 Location Transparent Communication

As with other systems, the runtime was extended to handle local or remote communication identically. No language extensions were required. The main runtime addition was a system *daemon actor* to demultiplex incoming discovery, adaptation, and communication requests, as well as indicate errors to the requesting actor, such as `CHANNEL_NOT_FOUND`. This actor is required given our zero-configuration approach.

4.3 Actor Adaptation

To support runtime adaptation, the system was extended to enable actor creation at a remote location (spawn), and moving an **executing** actor from one location to another (strong migration). This required the addition of two language statements, *spawn* (line 11) and *migrate*. The bulk of the work was focused on extending the runtime.

Spawn creates a new actor at a (local or remote) stage. Unlike a `new` operation (which is used to create new actors at a local stage), a spawned actor may be created with or without a reference to it. This either replicates the `new` operation in a distributed context, or offers a *fire and forget* option. The latter can be useful for factory actors.

Spawning an actor consists of transferring the class files of the actor, any types upon which it depends, and any initial state, to a specified stage where an instance of the actor is created. A connection is made to the stage using the information stored within the stage object passed to the spawn statement, otherwise, an exception is generated at the spawn call site, and the spawn is aborted. If successful, the classes upon which the actor depends, as determined by the compiler and described in the actor's class file, are encoded for transmission. Ensemble does not use on-demand loading of classes like Java, hence, all class files must be present before an actor is executed at a stage. This choice reflects the unreliable target environment, where devices are not always available and removes the need for class loading exceptions.

The constructor will then be invoked, and the actor created in a paused state. No type-checking is required at this point, as the compiler has already guaranteed that the constructor's arguments are legal. The remote stage will then communicate success to the actor at the spawn call site. Both local and remote actors will then continue execution. Otherwise, the remote stage will perform necessary garbage collection, and a relevant exception will be generated at the spawn call site. The set of potential exceptions are the same as a `new` operation, with the addition of the `StageNotReachableException`.

The above describes the process for a spawn *without* any reference. For a spawn *with* a reference, once the new actor has been created in a paused state, the remote stage will create a reference for it and its channels, and transmit these to the spawn call site, as in the discovery mechanism. Then, both actors will continue or a relevant exception is generated.

Migration entails an executing actor pause its execution, relocate to a specified stage, and then continue execution. As in a spawn, the actor's class files must be transmitted, but also the actor's state and current execution stack; this includes all channels, their connections, and buffer contents. This is an example of *strong* migration.

Once the transmission is received, the data objects, actor, and stack are reconstructed, including any reference counts. After the initial stage has received confirmation of successful migration, it will garbage collect the local actor, and tell the migrated actor to continue execution. Note that migration must maintain any channel connections between the actor being migrated and other actors. This is done after the actor has been reconstituted at the new stage. During migration, the actor is placed into the `MIGRATION` state to prevent channel operations reaching inconsistent states. In this state, communication actions are not eligible in this state, and the actor is not discoverable. This functionality is consistent with the existing Ensemble semantics. Once the actor has been reconstructed at the new stage, it is placed back into the `RUNNING` state, and recreates any connections which existed before the migration. Migration to the local stage does nothing. For any exceptional conditions, such as connection failure or lack of RAM on the new stage, migration is aborted, an exception raised at the migration call site, and allocated resources at the remote stage are released.

5 Evaluation

5.1 Applications

To motivate the case for runtime adaptation and understand the performance implications, four applications were used. Two are for parallel and embedded computations, and two are realistic applications described below.

Draughts - This application is a game with a computer opponent played via command-line interface. The application consists of two actor types, one to interact with the user (GUI), and one to calculate the computer player's moves (brain). There are two possibilities for runtime adaptation:

spawn - Before beginning its move, the computer player may decide to compute locally (easy mode), or remotely (hard mode). In the remote case, the computer player searches for a more powerful stage, and spawns an actor remotely, passing the current game board. Once the remote actor has explored the search tree, it transmits the choice back to the computer player. This style is similar to the use of RPC/RMI.

migrate - If the computer player decided to compute locally, but the search takes longer than some predefined time, it may choose to migrate the relevant actor(s) to a more powerful machine. Such stages may be discovered at startup, or on demand. Migration transparently enables the existing

work to be moved to another stage, rather than restarting or abandoning the search and returning the best result thus far.

In both approaches, the partitioning of the application into actors implicitly provides the necessary decoupling for either `spawn` or `migrate`. As the language supports these operations, their use is trivial from the user's perspective.

Mobile Media Player - This application displays a slide show of (ASCII-based) images on different hardware devices as a user moves through an area, and demonstrates adaptability in a mobile context. While not a modern media example, it represents the salient features of the application. The application has three actors. One to display images (*display*), one to access the file system at the starting stage (*file*), and one to locate physically close stages and instruct the display actor to either relocate, or return to the initial stage (*locator*).

This experiment consists of a laptop and three RPis, each in a different room. All actors begin on the laptop and images are displayed there. As the user enters a room, the application migrates the display actor to the RPi if within a certain distance. This distance is queried using the **#DISTANCE** property. After this point, the images are displayed on the RPi's output; the assumption being that the RPi's display is more appropriate. As the user leaves the room, the locator actor will instruct the display actor to migrate back to the laptop. If the user enters another room with a RPi, the actor will migrate directly from the previous RPi to the new one. All interaction (including file access) is performed via channels which transparently *stretch* as the display actor migrates.

5.2 Linguistic Complexity

```
1  stage home{
2  // opencl actor Multiply ...
3  actor Dispatch presents dispatchI{
4  query gpu_query(){$OPENCL==true and can_execute(Multiply);}
5  constructor() {}
6  behaviour {
7  // create the channels as normal ...
8  connect dout to i; connect o to din;
9  cfg = new config_t(ws,gs,i,o);
10 stages = findStages(gpu_query());
11 m = spawn Multiply() at stages[0];
12 connect requests to m.requests;
13 send cfg on requests; send d on dout;
14 receive result from din;
15 } }
16 boot{
17 d = new Dispatch();
18 <Remove actor creation + connection>
19 } }
```

Listing 1. Adaptable Matrix Multiplication

By using an actor-based model, creating distributed or adaptive applications is trivial from the developer's perspective. Listing 1 shows part of an Ensemble application which performs a matrix multiplication on a GPU via OpenCL. A full description is documented elsewhere [12]. The yellow highlighted sections indicate the changes required to find a remote GPU, and execute the kernel actor on it. No changes were made to the computational logic of the application.

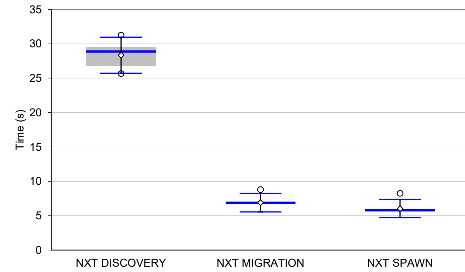

Figure 1. NXT Adaptation Performance Results

5.3 Performance Results

In the following, the hardware platforms referred to are described in Section 3, and the performance results are *turkey* box plots with 100 data points.

Resource Constrained Hardware Platforms Given the limited resources of the Tmote sky platform (10 KB of RAM), it was not possible to support our extensions. Instead, the NXT platform was used. This platform has 64 KB of RAM, which is four orders of magnitude less than the RPi which most other works target as an embedded platform. Figure 1 shows the time taken to discover a stage located on an NXT device, as well as spawn and migrate an actor here from a desktop machine across bluetooth. The actors used in these experiments contained a mixture of data types, including channels and arrays. Here, received data from a remote or local source is averaged, reducing the amount of data transmitted and power consumed. This application is representative of in-network data aggregation.

A limitation in the bluetooth chip on this particular platform prevented the use of the draughts or media application, as outgoing bluetooth connections could not be established. Note this is a limitation of this hardware platform, and not the runtime or programming model.

The large time seen for discovery is a combination of the (configurable) 7 second timeout for TCP-based connections, plus the 10.28 seconds required to discover all possible bluetooth devices. The remaining 11.07 seconds are required to connect to and communication with the stage on the NXT platform. In general, these times are larger by comparison to the other results in this section due to the use of bluetooth, which is unreliable and requires large buffering time periods, coupled with the NXT's slower hardware. 6.9 s and 6.0 s were required for migration and spawn, respectively.

Despite the slow performance, results show that it is possible to provide the functionality required for discovery and adaptation in a highly resource-constrained environment. Excluding some drivers, exactly the same runtime is used on this platform, as the others described in this section.

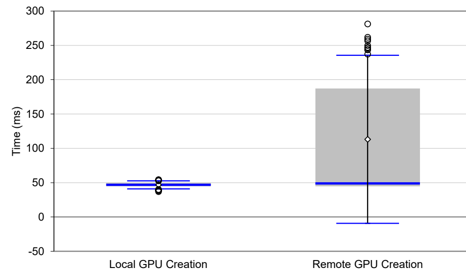

Figure 2. GPU Adaptation Performance

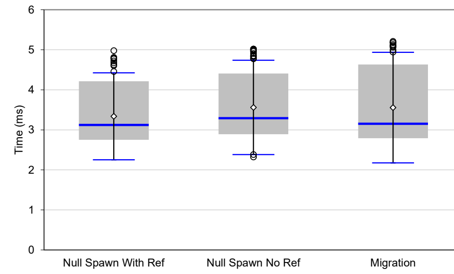

Figure 3. NULL Actor Adaptation Performance

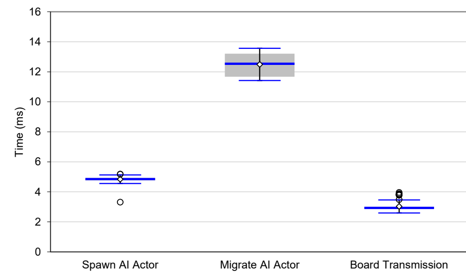

Figure 4. Draughts Adaptation Performance

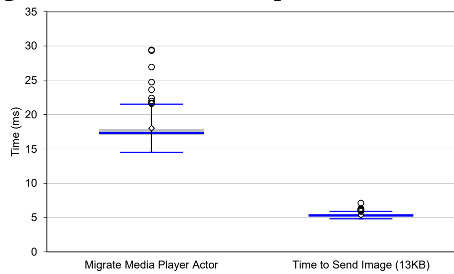

Figure 5. Media Adaptation Performance

Parallel Hardware Platforms As Ensemble abstracts OpenCL kernels as actors, adaptation can be applied here with no extra effort. The marshalling mechanisms were extended to handle the extra state required by OpenCL. Note that such actors may not migrate as the OpenCL runtime does not expose the necessary information. Figure 2 shows the time taken to spawn a kernel actor at a remote (GPU-enabled laptop) and local (desktop) stage connected by ethernet. On average, local creation takes 46 ms, and remote creation takes 113 ms. The large skew in remote creation is due to the OpenCL compiler, not the runtime. Although the cost of remote creation is (expectedly) higher than local creation, the average cost is likely to be significantly less than the benefit of using a parallel hardware architecture [12]. By constructing applications in this manner, stages without GPU support can easily take advantage of those which do. This topic has many promising avenues of future work.

Commodity Hardware Platforms

Null Actors - Figure 3 shows the base cost of adaptation via a *Null* actor - an actor with no channels or state, and the minimum amount of code to perform an adaptation. Actors were adapted from a desktop to a RPi via Ethernet. In the worst case, 204, 204, and 200 *Null* actors may be remotely spawned with a reference, spawned without a reference, or migrated per second, respectively.

Draughts -Figure 4 shows the time to spawn and migrate the brain actor from the RPi to the desktop via Ethernet. It also shows board transmission time, including the marshalling/demarshalling time at either stage.

A migration takes longer than a spawn as migration must capture the running state of the actor, as well as performing the spawn actions. The key point is that the potential benefits of offloading outweigh the cost of the operations. Given a search depth of three levels, the computer's move will be calculated on the order of seconds in the opening to mid stages of the game. The cost of adaptation is approximately 100 times faster than the time to complete a search, while the potential benefit is on the order of seconds, depending on the the migration stage target.

Media Player - For video playback, images are displayed at a rate of 24 frames/s (1/41.6 ms), giving the illusion of movement. Figure 5 shows the time to migrate the display actor between a desktop and a RPi via WiFi, including any referenced images. The worst case delay is 29 ms, with an average of 18 ms. This includes the time to reconstruct the actor, its state, and all channel connections. Figure 5 also shows the time taken to transmit the largest image (13 KB). Combined, the average migration (18 ms) and transmission time (5 ms) is lower than the 41.6 ms frame rate delay.

6 Conclusions and Future Work

Conclusion This paper demonstrates that an actor is the appropriate *unit of adaptation* to provide location transparent interaction and adaptation across a wide range of heterogeneous hardware platforms in a straightforward manner. This is shown by extending an actor-based framework to facilitate location transparent communication, runtime discovery, and adaptation, and is verified by experimentation.

Future Work As well as performing a comparative study with other technologies, there are three future avenues of investigate: replacement of executing actors, stage-based adaptation, and programmer guided, automated load-balancing.

References

[1] ANSA. 1989. *ANSA: An Engineer's Introduction to the Architecture.* Technical Report. Architecture Projects Managment Limited, Poseidon House, Castle Park, CAMBRIDGE, CB3 0RD, UK.

[2] Joe Armstrong. 2003. *Making Reliable Distributed Systems in the Presence of Software Errors.* Ph.D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[3] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. 1998. Mole : Concepts of a mobile agent system. *World Wide Web* 1, 3 (March 1998), 123–137. https://doi.org/10.1023/A:1019211714301

[4] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

[5] C. Cameron, P. Harvey, and J. Sventek. 2013. A Virtual Machine for the Insense Language. In *MOBILe Wireless MiddleWARE, Operating Systems and Applications (Mobilware), 2013 International Conference on.* 1–10. https://doi.org/10.1109/Mobilware.2013.17

[6] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. 2013. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* ACM, New York, NY, USA.

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05).* USENIX Association, Berkeley, CA, USA, 273–286. http://dl.acm.org/citation.cfm?id=1251203.1251223

[8] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. Knoxville, TN, USA.

[9] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso. 2009. Calling the Cloud: Enabling Mobile Phones As Interfaces to Cloud Applications. In *Proceedings of the ACM/IFIP/USENIX 10th Intl Conf on Middleware (Middleware'09).* Springer-Verlag, Berlin, Heidelberg, 83–102. http://dl.acm.org/citation.cfm?id=1813355.1813362

[10] Philipp Haller and Martin Odersky. 2006. *Modular Programming Languages: 7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13-15, 2006 Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Event-Based Programming Without Inversion of Control, 4–22. https://doi.org/10.1007/11860990_2

[11] Paul Harvey. 2015. *A linguistic approach to concurrent, distributed, and adaptive programming across heterogeneous platforms.* Ph.D. Dissertation. School of Computing Science, University of Glasgow. http://theses.gla.ac.uk/6749/

[12] Paul Harvey, Kristian Hentschel, and Joseph Sventek. 2015. Parallel Programming in Actor-Based Applications via OpenCL. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15).* ACM, New York, NY, USA, 162–172. https://doi.org/10.1145/2814576.2814732

[13] Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen. 2009. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In *Proc of the 7th Intl Conf on Advances in Mobile Computing and Multimedia (MoMM '09).* ACM, New York, NY, USA, 195–203. https://doi.org/10.1145/1821748.1821787

[14] Jonathan W. Hui and David Culler. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc of the 2nd Intl Conf, Embedded networked sensor sys (SenSys '04).* ACM, New York, NY, USA, 81–94. https://doi.org/10.1145/1031495.1031506

[15] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 109–133. https://doi.org/10.1145/35037.42182

[16] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. 2010. Turning down the LAMP: software specialisation for the cloud. In *Proc of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10).* USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1863103.1863114

[17] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. 2000. Process migration. *ACM Comput. Surv.* 32, 3 (Sept. 2000), 241–299. https://doi.org/10.1145/367701.367728

[18] Jan S. Rellermeyer, Oriana Riva, and Gustavo Alonso. 2008. AlfredO: an architecture for flexible interaction with electronic devices. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08).* Springer-Verlag New York, Inc., New York, NY, USA, 22–41. http://dl.acm.org/citation.cfm?id=1496950.1496953

[19] Z. Sanaei, S. Abolfazli, A. Gani, and R. Buyya. 2014. Heterogeneity in Mobile Cloud Computing: Taxonomy and Open Challenges. *Communications Surveys Tutorials, IEEE* 16, 1 (First 2014), 369–392. https://doi.org/10.1109/SURV.2013.050113.00090

[20] Carlos Varela and Gul Agha. 2001. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.* 36 (December 2001), 20–34. Issue 12. https://doi.org/10.1145/583960.583964