Position Paper: Elastic Processing and Storage at the Edge of the Cloud

Steffen Viken Valvåg Dag Johansen Åge Kvalnes
Department of Computer Science
University of Tromsø, Norway
{steffenv,dag,aage}@cs.uit.no

ABSTRACT

Cloud services traditionally have a centralized architecture, where all clients communicate individually with the central service, and not directly with each other. Data is primarily stored in the cloud, and computations that touch data are performed in the cloud. We present Rusta, a platform that allows cloud services to deploy in a more flexible and decentralized manner, potentially involving the client machines at the edge of the cloud both for storage and processing of data. This can reduce operational costs both by leveraging freely available client resources, and by reducing data traffic to and from the cloud.

Rusta includes a group abstraction to delineate webs of trusted peers, a light-weight process abstraction based on asynchronous message passing, and a distributed data storage layer. For elasticity, processes may migrate freely among the clients of a group, and can be replicated in a transparent manner. A central hub service executes in the cloud and maintains critical system state, while delegating work to clients as appropriate. This paper describes the design and current implementation of Rusta, its high-level programming model, and some of its potential applications, in particular as a foundation for highly elastic computations at the edge of the cloud.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

Keywords

Rusta, Light-weight processes, Migration

1. INTRODUCTION

Services deployed in the cloud commonly enjoy great scalability, elasticity, and fault tolerance. Computations traditionally performed on client devices and machines are increasingly offloaded to services hosted in the cloud. In many cases, such offloading is desirable, for example when battery-powered devices wish to conserve energy. However, client machines may also represent an untapped computing resource. If some work can be shifted from the cloud-hosted machines to client machines, whose resources are generally little utilized and freely available, operational costs can be reduced. Similarly, if data traffic can be routed directly between clients instead of going through the cloud, the savings can be significant. In some scenarios, for example when integrating data from multiple social networking applications, the relevant data is also locked in with other cloud services and only available at the clients.

This paper presents our ongoing work with Rusta[1], a generic platform for decentralized, distributed computations that utilize resources both on client machines and in the cloud. The platform aims to simplify development of applications that largely execute at the edge of the cloud, integrating local data from client machines, while also drawing on resources in the cloud whenever the need arises.

One potential obstacle to this vision is trust. Some users may already be reluctant to trust the integrity and confidentiality of their data to a cloud service. Trusting their data to be stored locally on random peers is certainly even less attractive. To address this concern, Rusta exposes a fundamental group abstraction as the unit of trust. Clients may be members of multiple groups, and while data and computations are confined to specific trusted groups, they may migrate freely within them. This preserves elasticity while clearly delineating varying levels of trust.

The group abstraction is a particularly good fit for social applications, which are inherently about establishing networks of trust between end-users. If group membership is bootstrapped from social network data, users will only be trusting their data to real-life friends, as opposed to random strangers. Clients also play an active role in social applications, contributing and generating content instead of just passively consuming it. As such, a decentralized approach to analytics for social applications has a particular appeal, and Rusta aims to facilitate it.

The rest of this paper is structured as follows. Section 2 gives a high-level view of Rusta and its architecture. The programming model and interfaces that Rusta applications relate to are described in Section 3. Section 4 covers lower-level implementation details. Section 5 discusses potential Rusta applications, focusing in particular on an image shar-

[1]In the Sami language, *Rusta* is a local winter fog.

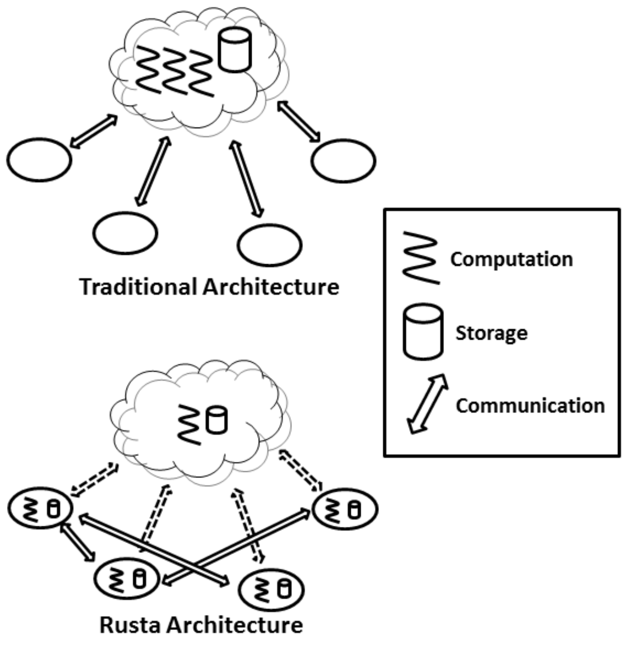

Figure 1: Rusta Architecture, compared to the traditional architecture for cloud services.

ing application, to give a more complete understanding of how to apply Rusta. We discuss related work in Section 6, and conclude in Section 7.

2. ARCHITECTURE

Rusta draws on insights from our previous work in the area of big data analytics. With Cogset [11] we explored new mechanisms for routing and placement of data in a MapReduce engine. In contrast to conventional MapReduce engine designs, Cogset employed predetermined data routing schemes to avoid the need for temporary copies of intermediate data. This combination of tight coupling of storage and processing and a functional style of programming resulted in better data locality, and, as a consequence, significant performance improvements.

With Rusta, our focus has shifted to analytics that may be performed in real-time, potentially on widely distributed data only found at the edge of the cloud. Elements of Cogset's design remain, by focusing on a functional style of programming that maintains location transparency, while allowing the underlying run-time to arrange for the desired data locality. At the same time, Rusta includes support for light-weight asynchronous message processing to cater for real-time processing needs.

Traditional cloud services have a centralized architecture, where all clients communicate individually with the central cloud service, and never directly with each other. Data is stored in the cloud, and computations that touch data are performed in the cloud. Clients are generally simple and stateless, and frequently just web browsers accessing the service through a web interface. Application developers write the code that executes in the cloud, within the framework of the chosen cloud provider. Common abstractions that developers must relate to include HTTP request handlers, daemon processes for background activities, main memory key/value caches, and various kinds of persistent storage.

Rusta decouples the application logic both from the specific cloud provider and from the exact mode of deployment. At one end of the spectrum, applications may execute in a very decentralized manner, only using the cloud for occasional coordination, while storing data and executing code locally on client machines. They may also fall back to the traditional architecture at the other end of the spectrum, relying on the cloud both for storage and computations, as dictated by load or other circumstances.

Figure 1 shows a high-level view of how Rusta's architecture differs from the traditional architecture. Computations may execute on clients as well as in the cloud, and data may also reside locally on clients, and be accessed directly from other clients. As a consequence, the amount of communication between the cloud and its clients is reduced, and instead some of the communication goes directly from one client to another.

In large systems, every client should not necessarily be exposed to every other client. As such, a central abstraction in Rusta is the group. A client may create a new group, which will initially contain only itself. It may then invite other clients to join its group, and accept invitations from others, expanding its set of group memberships. Groups may also include members in the cloud, when cloud resources are integrated in the more traditional way.

Groups are containers for processes and data. Each group defines a separate namespace, so process A in group X is different from process A in group Y, and similarly for data items. Processes are simple message processing loops, and may migrate freely among the members of their group. Data is accessible to all processes in a group, and stored with a certain replication degree within it. Cloud storage may also be employed as a backup, for increased reliability.

Individual clients may come and go during the lifetime of a Rusta application. Continuity and progress is ensured by the *hub*, which is a central component implemented as a conventional cloud service. The hub maintains all group membership and namespace information, and may arrange to migrate processes or data between client machines. It is the unique authority on the state of the system, simplifying many practical aspects.

Being a conventional cloud service, the hub can generally be assumed to be available, reliable, and scalable. As such, Rusta enjoys the architectural simplicity of having a single centralized component, but avoids the usual drawback of having a single point of failure. So long as client failures can be tolerated, the risk of failure from Rusta is no greater than the risk of failure from a service hosted fully in the cloud.

Data transfers may also go directly between clients; the hub needs only be involved in namespace lookups and modifications. In that respect, the hub is analogous to the master node of distributed file systems like GFS [4], or to a BitTorrent tracker, albeit implemented as a reliable cloud service and not as an unreliable singleton component.

Rusta may also be deployed in a more traditional manner, utilizing the cloud for more than just hosting the hub service. In addition to the client machines of end-users; virtual machines provisioned from the cloud may also take the role of clients, adding additional computational resources on de-

```
object ImageService extends Client {
  group("images") {
    process("main") {
      case 'start => {
        send("thumbnails", ('get, "image.jpg")) {
          case null => println("No such image")
          case tn: Image => showImage(tn)
        }
      }
    }

    process("thumbnails") {
      case ('get, path: String) => {
        getData(path) {
          case (image: Image, thumbnail) => {
            reply(thumbnail)
          }
          case (image: Image, null) => {
            val tn = makeThumbnail(image)
            putData(path, (image, tn))
            reply(tn)
          }
          case null => reply(null)
        }
      }
    }
  }
}
```

Figure 2: Example Rusta client.

mand. The hub service does not discriminate between the two cases. Similarly, while our design is focused on providing distributed data storage that can exploit local storage on clients, it is possible to complement this with storage in the cloud, as additional back-up.

3. PROGRAMMING MODEL

Rusta is implemented in Scala, a powerful multi-paradigm language that integrates features of both object-oriented and functional languages. Scala programs are compiled for the Java Virtual Machine and interface seamlessly with Java code, but have a number of additional features, such as type inference, pattern matching, and closures, at their disposal. With its flexible syntax, Scala is particularly well suited for embedding of domain-specific languages.

Rusta processes are message processing loops expressed directly as Scala code. Incoming messages are demultiplexed in a highly expressive and efficient way using the native pattern matching features of Scala. In response to a message, a process can for example modify its internal state, dispatch asynchronous messages to other processes, or access or modify persistent data.

Figure 2 shows an example Rusta client, illustrating how parts of a distributed image sharing service might be implemented using Rusta. The client creates two processes named "main" and "thumbnails", contained in a group called "images". The main process sends a message to the thumbnails process, requesting a thumbnail representation of a particular image. The thumbnail process accesses data storage, looking up the thumbnail if it exists, or creating and storing it if not. The thumbnail is then returned to the main process that requested it.

The example code features what appears to be syntactical blocks introduced by the *group* and *process* keywords. In reality, these are not keywords but plain functions, and the bracketed blocks that follow are closures passed as (curried) arguments to the functions. In the first case, when the *group* function is invoked, the associated closure is evaluated to populate a certain part of the hierarchical group namespace. The group name "images" is pushed onto a stack, and when processes are subsequently created, their location in the namespace is determined by that stack.

Similar techniques are used throughout Rusta's client library, to give the programming interface the appearance of a domain-specific language that extends Scala with special syntax. In reality, it is just a library that exploits the powerful features of Scala to create useful abstractions.

3.1 Processes

In the case of processes, the closure passed to the *process* function is a *partial function* that defines its behavior. The partial function is the main message handler of the process and uses pattern matching to specify how incoming messages should be processed. Each message is matched programmatically against each of the *case* expressions. The main process in the example reacts to a single system-generated *'start* message. (Literals like *'start* are *symbols*, which correspond to interned Java strings.) In response to the *'start* message, the process then sends a message to the thumbnails process, requesting a thumbnail representation for a particular image. In this case, the request is a tuple with a symbol denoting the kind of request (*'get*) and the path of the image in question. However, this is just a convention used by the particular application; messages can be arbitrarily complex objects.

When sending messages, a *reply handler*—i.e., a specification of how to process the eventual reply—can be given in another partial function directly following the *send*. This is again a closure, passed as a curried argument to the *send* function. The reply handler is stored by Rusta and only invoked later, if or when a reply arrives. Outgoing messages are automatically tagged with identifiers, and a mapping is maintained to determine which reply handler to invoke for any incoming messages. Messages that are not replies to previous requests are processed by the main message handler, i.e. the outermost block of the process. Thus, the act of sending a message and processing the corresponding reply has the appearance of a synchronous operation, but is in reality fully asynchronous. In the example, the expected reply from the thumbnails process is either *null*, indicating an unknown image, or an actual *Image* object — i.e., the requested thumbnail. Messages that do not match any message handlers are discarded. (They are also logged, for debugging purposes, since they may be symptoms of bugs in the application.)

3.2 Data Access

Data is accessed in a functional style that resembles communication between processes. The example in Figure 2 performs some data manipulation in the thumbnails process, using the *getData* and *putData* functions. The *getData* function looks up the persisted value associated with a given key, and invokes the supplied closure with the given value. Values are matched against the sequence of *case* expressions in a corresponding manner to how incoming messages are demultiplexed. In the example, the value associated with

```
object StatefulExample extends Client {
  group("example") {
    var x = 0

    process("counter") {
      case 'increment => {
        x += 1
      }
      case 'get => {
        reply(x)
      }
    }
  }
}
```

Figure 3: Example stateful process.

an image path is either *null*, in which case a *null* reply is sent, or it is a tuple with the image data and its associated thumbnail data. If the image exists, the thumbnail has either been generated previously, or the thumbnail data is null, and must be generated and stored using *putData*. The pattern matching features of Scala offers a clear and concise way to express the three different possibilities, and the corresponding actions to take.

It should also be noted that *getData* is a fully asynchronous operation. While the library design makes it appear as a synchronous operation, the closure that matches data values and specifies what to do with them can in fact be evaluated asynchronously. As such, the closure can be stored until the data in question is available and only invoked then, allowing the originating process to make progress in the meantime by processing other messages. In fact, the closure could well be transmitted over the network to a different client before it is evaluated, as described next.

3.3 Stateful Processes

When processing data streams, such as sequences of messages, aggregation is usually a core operation. This requires stateful message processing. Similarly, routing of messages may rely on stateful routers. Internal state can also be used to optimize otherwise stateless processes, e.g. through caching of intermediate results. In short, there are a range of use cases where a simple way to maintain internal process state is necessary or highly beneficial.

Rusta meets this requirement by allowing processes to maintain internal state in the form of regular variables. This is achieved by having the main message handler close over variables defined in an outer scope. Since messages are processed sequentially, the addition of internal state does not complicate process logic with concerns such as race conditions. Figure 3 shows an example of a stateful Rusta process. In the example, the partial function that defines the "counter" process closes over the variable *x*, defined in an outer scope. This offers a much more convenient alternative to using the asynchronous *getData* and *putData* API. Going through the persistent storage layer just to maintain internal state of this kind would also be highly inefficient.

4. IMPLEMENTATION

A key idea behind Rusta is to maximize location transparency. As such, processes should be able to migrate between the clients of a group, and checkpoint their state to the central hub service. When accessing data using *getData*, the system should be free to choose an execution strategy; either the relevant data can be copied to the client where the requesting process executes, or the process can be migrated to a client where a local replica of the data already exists. The latter approach may be much more efficient in cases where large data items are accessed.

If processes were implemented as threads, migration would be fairly complex to implement, involve the copying of both register state and stack contents, and require underlying support at the virtual machine level [5]. However, Rusta is designed around closures as the unit of execution. Messages are processed by invoking a closure, and with normal usage, a message handler will never block. As noted in Section 3, sends and data accesses may appear to be synchronous operations, but are in reality executed asynchronously, based on closures that specify future work. In effect, Rusta programs are continuation-based under the hood, but rely on syntactic sugar to mimic a more imperative programming style, which is more familiar to most programmers.

Instead of tying each process to a specific thread, Rusta programs execute by scheduling very fine-grained tasks on a thread pool, where a typical task is the processing of one message, or the matching of one data item (i.e. the completion of a *getData* call). The state of a process can therefore be captured quite easily, and at regular intervals, after the processing of each message. It is just a matter of serializing its state, including all associated closures. So while Rusta processes superficially resemble threads, they are in fact represented as regular objects living on the heap, and fully decoupled from any underlying threads.

This paves the way both for checkpointing and migration of processes, implemented at the "user level", i.e., with no support from the virtual machine. Processes are also light weight, with a relatively small memory footprint, and not constrained by the number of threads that can be created. Thus, they may be numerous; applications are free to create thousands of processes without worrying about overhead, which may be useful to simplify application design. Inactive processes may also be "hibernated" by storing their state in the hub service, and leaving them dormant until an incoming message arrives. The trade-off with that approach is higher latency when processing that message, as the process must first be re-activated at one of the clients.

Rusta is still a work in progress, and many details of its implementation remain in flux. However, the major components and most important functionality is implemented and working. The hub service executes on Google App Engine (GAE), in its Java servlet environment. The choice of cloud provider for this service is arbitrary, and we don't anticipate much difficulty with porting the hub service to other cloud providers, if desired. The main requirement for the hub service is to have high availability, since it is the central component of our architecture. By trusting the cloud to provide a highly available hub service, we simplify other aspects of our architecture. For scalability, the hub service relies on the inherent capabilities of GAE to dynamically scale out depending on load.

Clients access Rusta through the client Scala library, which offers the high-level programming abstractions described in Section 3. The library communicates with the hub service using RPC calls (over HTTP), and also maintains an internal actor system for each client using the Akka actor library. Clients communicate with each other using Akka's remoting support, with Netty, an asynchronous I/O library, as the transport layer. Clients have a local directory for storing replicas of data items, and a public/private key pair for identification. They register with the hub service upon start-up, and communicate regularly with it to obtain scheduling decisions (i.e., which processes to execute) and data transfer instructions.

The hub service maintains various data structures in the GAE datastore. A global class repository contains all the Java classes involved in the application, i.e. all of the application's code[2]. Since both process migration and data access rely on transferring serialized objects, all clients must have a common view of the application's classes, to remain compatible. An interesting sub-problem which we intend to investigate in the future is how to handle code upgrades, to allow long-running Rusta applications to upgrade in place while running. For large scale applications that involve numerous clients, this may be the only realistic way of upgrading at all. For now, the foundations for such mechanisms are in place, by making sure that clients load classes from the hub, and not from their local class path.

The hub also stores the hierarchical group name space, including both the state of all processes, and the meta-data for all data items. The state of a process includes the client where it currently executes (if any), and its last checkpointed internal state. The meta-data associated with data items most crucially includes a list of the clients hosting its replicas. The algorithms for allocating replicas and for checkpointing processes are currently rudimentary, and we expect to focus much of our future work on these areas. While it is possible to delegate many issues of consistency to the application level, the aim is to minimize the burden placed on application developers, and offer consistent checkpointing of whole sets of cooperating processes.

5. APPLICATIONS

As noted in Section 2, Rusta is motivated by and founded in extensive previous experience with big data analytics. Our work is also part of a larger research project focusing on new directions for information access systems. To further validate our design choices, we are in the process of developing a set of applications drawn from this domain. Applications include among others Lifelog image analysis to produce daily activity summaries [6], personal sensor data processing and storage in the soccer domain [7, 8], and transparent data replication across clouds [9].

These applications currently run on different platforms and systems, and have diverse internal architectures. By transferring and adapting them to run on Rusta we hope to gain experience with a broad spectrum of Rusta usage. For example, the Lifelog application will stress how Rusta processes can transparently draw on cloud resources to accommodate computationally weak client devices. The personal sensor data application will yield experience with handling sensitive and private data. The cloud replication application will stress Rusta multi-cloud deployments.

Rusta currently only supports single-hub Google App Engine deployments. Running on a different cloud would entail porting the hub service code to the new cloud platform (which is straightforward), but we are currently undecided about how to approach multi-cloud deployments. One approach would be to use a multi-hub setup with hubs communicating among themselves to maintain consistent state. An alternative, and appealing, approach is to keep the simplicity of a single hub service, and model additional clouds as (trusted) Rusta clients that could be invited to a group for members to exploit their potentially considerable computational resources. The Lifelog application will use this approach, with the application user owning multiple clients that Rusta can transparently exploit for additional processing resources.

In the following we outline one application in more detail; an image sharing application. The application intends to investigate use of Rusta to build an image sharing system with features similar to services like Instagram, only with a *decentralized* design. A design goal is to ensure that the images proper can be communicated among clients without having to rely on storage in a centralized cloud service.

The application uses the Rusta group abstraction to create a container and medium for sharing images. A group is initially created by a user that wishes to share some images with a select set of other users. The members of the group could for example be bootstrapped from the user's Facebook friends list, or some other source of social connections. Crucially, the group creator presents to the hub a set of public keys that identifies group members. Recall from Section 4 that a client registers with the hub when starting up. Part of this registration is presenting the hub with a public key that corresponds to the user owning the client. The hub uses this key to determine pending group invites.

Recall that accepting group membership also implies willingness to host group processes and data on the client. When a user accepts group membership (and having passed the appropriate key authenticity checks), the image sharing application creates one Rusta process that will act as the *steward* of the user within the group. The code for this process can either be supplied by the image viewing software that interfaces with Rusta on the client device, or it can be supplied by the steward of the creator of the group. Since the steward is a Rusta process it can run on any client available to the group and it can rely on Rusta services to checkpoint state. The image sharing application requests, upon group creation, that stewards preferably run on the clients of their owners. When no such client is online, stewards run on other clients in the group. Assuming a client disconnects from Rusta properly, hosted processes are migrated before the disconnect, and it can be assumed that their state is checkpointed.

When the creator of the image sharing group writes a new image to the group, a thumbnail of the image is created and a message containing the thumbnail is sent to all stewards. Stewards are stateful processes that wait for such messages and aggregate them internally. When users log on to the image sharing application, they may retrieve a list of new images by requesting it from their steward. Example code for the steward processes is listed in Figure 4. In effect, stewards act as synchronization points for individual users.

When creating the image sharing group, the creator requests a replication degree for group data. What clients should hold an image replica is decided by the hub. At any given time, some clients will be offline while others will be online. It is the responsibility of the hub to implement poli-

[2]Recall that Scala code also compiles to Java class files.

```
object ImageSharing extends Client {
  group("image-sharing") {
    var thumbnails: List[(Image, String)] = Nil

    process("steward-"+userName) {
      case ('newImage, tn: Image, path: String) => {
        thumbnails ::= (tn, path)
      }
      case ('getNewImages) => {
        val msg = thumbnails
        thumbnails = Nil
        reply(msg)
      }
    }
  }
}
```

Figure 4: Example steward process for a user in an image sharing service.

cies that improve group data availability. For example, if some clients are more frequently online than others, those clients are likely to be more aggressively used for replicas than other clients. The Rusta hub will never store group data, it will only mediate and direct transfer of group data among clients. Thus, for the image sharing application, it may take some time for the desired replication degree to be reached; a sufficient number of clients must be online for a duration long enough to complete replication. Note that the Rusta hub will store messages sent to a group process before relay. Thus, in the image sharing application, thumbnail messages will be exposed to the cloud. We considered this to be an acceptable cloud exposure, for the application to provide a useful image notification mechanism.

Proper image availability, however, is dependent upon client availability. If a user decides to retrieve an image, a client with a replica must be online. If no such client is available, the user has to wait for a particular client to come on-line. Thus, data availability will increase with the number of group members. This is a fundamental trade-off with only relying on clients for data replication. Another issue is trust. For the image sharing application to avoid cloud exposure, group members must be trusted not to introduce cloud-based clients.

6. RELATED WORK

There are three main ideas in Rusta: Take advantage of resources at the edge of the cloud, access data in a functional style to encourage data locality while preserving location transparency, and provide a high-level distributed programming model with a unified system view to resources for both computing or storage.

Of previous work, Eyo [10] is perhaps the system that most closely matches our vision, with its focus on providing device transparency. In Eyo, users are given a single view of all their files, regardless of which device they are using to access them. This device transparency is a stronger property than location transparency, in that it implies a consistent view even under disconnected, off-line operation. In Rusta, we choose to rely on the hub service for availability, and focus more on providing efficient and expressive mechanisms for on-line applications. To clarify the distinction, Eyo is a system that could feasibly be implemented using Rusta as the underlying platform.

As noted in Section 4, Rusta supports migration and checkpointing of processes without relying on support from the operating system or the virtual machine. This is achieved through a continuation-based implementation that revolves around closures that are evaluated asynchronously. At the same time, the exposed programming abstractions enable a familiar imperative programming style that resembles thread-based concurrency.

Off-loading of work from client devices is a hot topic [1, 3], and thread migration can be used as the underlying mechanism for this, such as in the recent COMET [5] system. To realize thread migration, COMET made substantial changes to the Dalvik virtual machine, both to trace modified objects and to insert synchronization points. Rusta implements migration purely as a library and thus offers a compelling alternative.

The idea of utilizing available resources on the machines of end-users can be traced back at least to applications like SETI@home, Folding@home, and Freenet. Building on that, TFS [2], a file system for contributory storage, is an example of a system that builds more generic abstractions on top of client machine resources. AmazingStore [12] complements centralized cloud storage with edge-based replicas to improve availability and protect against correlated failures. These systems are constrained to storage, while Rusta also considers how to implement processes that live in a global namespace.

7. CONCLUDING REMARKS

Rusta is a platform that allows cloud services to deploy in a flexible and decentralized manner, taking advantage of resources made available by its clients. By accessing data in a functional style, Rusta encourages local data access while preserving location transparency. Rusta offers a hierarchical namespace for groups that delineate sets of mutually trusted clients. The namespace also presents applications with a single, unified view of all processes and data. Processes can be stateful, but are designed such that Rusta can transparently capture and checkpoint their state.

We are currently working on refactoring several existing cloud services to use Rusta as their execution platform. In particular, we are focusing on highly decentralized social networking services that offer traditional functionality, but are able to exploit idle resources at the client machines as a new strategy for scaling out. Our goal is to enable highly elastic and flexible computations that execute efficiently and cheaply at the edge of the cloud.

8. ACKNOWLEDGMENTS

This work is part of the iAD project[3] and partially funded for a period of 8 years by the Research Council of Norway as a Centre for Research-based Innovation.

The iAD project is directed by Microsoft Development Centre Norway, in collaboration with Accenture, Cornell University, Dublin City University, BI Norwegian School of Management, and the universities in Oslo (UiO), Trondheim (NTNU), and Tromsø (UiT).

[3]http://www.iad-centre.no

9. REFERENCES

[1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[2] J. Cipar, M. D. Corner, and E. D. Berger. Tfs: a transparent file system for contributory storage. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, pages 28–28, Berkeley, CA, USA, 2007. USENIX Association.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[5] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: code offload by migrating execution transparently. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[6] C. Gurrin, T. Aarflot, and D. Johansen. Gardi: A self-regulating framework for digital libraries. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 1, pages 305 –310, oct. 2009.

[7] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen. Bagadus: An integrated system for arena sports analytics - a soccer case study. In *Proceedings of the 4th ACM International Conference on Multimedia Systems (MMSys)*, Oslo, Norway, To appear March 2013. ACM.

[8] D. Johansen, M. Stenhaug, R. Hansen, A. Christensen, and P.-M. Hogmo. Muithu: Smaller footprint, potentially larger imprint. In *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pages 205 –214, aug. 2012.

[9] A. Nordal, A. Kvalnes, J. Hurley, and D. Johansen. Balava: Federating private and public clouds. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 569 –577, july 2011.

[10] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and F. Kaashoek. Eyo: Device-transparent personal storage. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX '11)*, Portland, Oregon, June 2011.

[11] S. V. Valvåg, D. Johansen, and A. Kvalnes. Cogset: a high performance mapreduce engine. *Concurrency and Computation: Practice and Experience*, 25(1):2–23, 2013.

[12] Z. Yang, B. Y. Zhao, Y. Xing, S. Ding, F. Xiao, and Y. Dai. Amazingstore: available, low-cost online storage service using cloudlets. In *Proceedings of the 9th international conference on Peer-to-peer systems*, IPTPS'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.