Submitted for publication. Contact larus@microsoft.com for citation information.

Orleans: A Framework for Cloud Computing

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, Jorgen Thelin
eXtreme Computing Group, Microsoft Research

Abstract

Client + cloud computing is a disruptive, new computing platform, combining diverse client devices – PCs, smartphones, sensors, and single-function and embedded devices – with the unlimited, on-demand computation and data storage offered by cloud computing services such as Amazon's AWS or Microsoft's Windows Azure. As with every advance in computing, programming is a fundamental challenge as client + cloud computing combines many difficult aspects of software development.

Orleans is a software framework for building client + cloud applications. Orleans encourages use of simple concurrency patterns that are easy to understand and implement correctly, building on an actor-like model with declarative specification of persistence, replication, and consistency and using lightweight transactions to support the development of reliable and scalable client + cloud software.

1. Introduction

Client + cloud computing is a disruptive, new computing platform, combining diverse client devices – PCs, smartphones, sensors, and single-function and embedded devices – with the unlimited, on-demand computation and data storage offered by cloud computing services such as Amazon's AWS or Microsoft's Windows Azure. Advances in semiconductors again are driving a radical change, reducing the cost of computing and communications and enabling inexpensive, compact, personal, and mobile devices with powerful processors, wireless connectivity with good bandwidth and reach, and low power consumption. In the data center, low-cost, efficient, virtualized servers created a new business of selling inexpensive computation on demand. Together these advances make possible the vision of ubiquitous computing articulated by Mark Weiser two decades ago [1], where data and computation are no longer tied to a physical location or computing device, but instead become the fabric of our environment and part of all devices we interact with.

As with every advance in computing, programming is a fundamental challenge. Client + cloud computing combines many of the most difficult aspects of programming. These systems are inherently parallel and distributed, running computations across a large number of servers in multiple data centers and diverse clients. Individual computers and communication links are commodity components, with non-negligible failure rates and complex failure modes. Cloud computing runs as a service, offering economies of scale and efficiency by concurrently processing requests from many clients, but also facing challenging demands in handling varying and unpredictable loads and offering a highly available and reliable service in the face of hardware and software failures and evolution. These problems, of course, come in addition to the familiar challenges of constructing secure, reliable, and efficient software.

Orleans is a software framework for building cloud applications. It offers a simple programming model built around grains, logical units of computation with private state that communicate exclusively by sending messages. At any time, a grain may have zero, one, or multiple activations, the physical instantiation of a grain on a server. To simplify programming and reduce the opportunity for errors, the programming model imposes restrictions on concurrency. A grain activation processes an external request to completion before turning to the subsequent request. The system as a whole can process multiple requests concurrently in distinct grain activations, but these computations are isolated, except at the clearly identified points where grains commit state changes to persistent storage and make them globally visible. This is far more restrictive than most shared memory or message-passing models, but still flexible enough to build general-purpose applications.

A key challenge in building these applications is scale. A successful and popular application must handle demand that grows orders of magnitude in a very short time. Software, like the cloud hardware platform, should be elastic and capable of expanding to meet demand. Growth not only increases the number of servers and network connections, and consequently the likelihood of failures, but it also increases the size of data structures and the cost of algorithms. Techniques suitable for small websites, such as centralized databases, become bottlenecks as systems grow and must be replaced by more scalable solutions, such as sharded databases [2], often requiring architectural changes and rewrites of a system [3]. Orleans's grain model addresses these concerns. Grains encourage a sharded style of computation, with small, independent computations distributed across servers to closely match the semantics of scalable, sharded databases. Grains can be inexpensively migrated or replicated among computers, even while a service is running, to balance load and to reduce communication cost by moving computation to data or co-locating communicating entities.

In addition, a shared service must be both available and reliable. The more popular a service, the higher the demand for availability, although greater scale increases the difficulty of achieving this goal. Distributed systems research has produced many techniques for building reliable systems from unreliable building blocks. The general approach is to replicate a computation or datum, and if one copy fails, the others will service requests while the failed copy is restarted or reconstructed. Multiple, concurrently updated copies introduce the need to ensure that distinct computations provide a single, consistent result, regardless of which copy is accessed. Orleans grains separate the application's logic from its replication and consistency mechanisms. The Orleans runtime can replicate a grain's state and offers predictable consistency guarantees for the copies.

This paper makes the following contributions:

- Identifies the challenges in building client + cloud software.

- Describes a solution to these problems based on concurrent, replicated, asynchronous units of computation.
- Extends the basic solution with light-weight transactions that ensure isolation, consistency, and error recovery.
- Integrates persistence, replication, and consistency into the programming model and supports it in the language runtime.
- Shows how a runtime can improve an application's performance by transparently distributing computations among servers.

The rest of the paper is organized as follows. Section 2 provides an overview of Orleans while section 3 describes the programming model in more detail. Section 4 briefly discusses the implementation. Section 5 describes sample applications. Section 6 presents performance measurements, and section 7 surveys related work.

2. Orleans Overview

Orleans supports the development of cloud application software. It targets a specific category of software, embodying best practices for building scalable cloud applications. This section provides an overview of Orleans, elaborated in later sections, and explains the rationale for the design choices.

2.1 Grains

In Orleans, **grains** are atomic units of isolation, distribution, and durability. A service is constructed from grains running on servers in a data center. An external request, from a client of the service, is sent to a grain for processing. A grain can concurrently invoke the operations of other grains through asynchronous messages. Grains internally are not parallel; they process a request fully before handling the next one.

Multiple instantiations of a grain, known as **activations,** process multiple independent requests to a service in parallel. Orleans creates multiple activations of a grain to handle simultaneous requests, increasing system throughput, reducing queuing latency, and improving system scalability. An activation is completely isolated from other activations of the same grain and only interacts with different grains' activations by exchanging messages. Activations cannot share memory or invoke each other's methods directly.

The state of a grain is persistent by default, so Orleans transfers modified grain state from server memory to durable storage, and vice-versa, without explicit application code (Section 4.2). A grain may exist only in the persistent store – i.e., the grain is not active on any server – when no requests for the grain are pending. When a request arrives, the Orleans runtime activates the grain by choosing a server, instantiating an activation that implements the behavior of the grain, and initializing it with the grain's persistent state. If the grain modifies its persistent state, the runtime updates persistent storage with the in-memory version at the completion of the application's transaction.

Multiple activations may concurrently modify a grain's state, requiring a mechanism to reconcile changes to their shared, persistent state. Orleans uses a multi-master, branch-and-merge update model, similar to Burckhard's revision-based model [4].

Orleans allows grain activations to be placed independently on any server in a system and to be migrated between servers in order to balance load, increase failure tolerance, or reduce communication overhead. As a consequence, Orleans tracks the location of each activation in a distributed directory, which provides the flexibility to place and migrate grains dynamically. The directory may grow to millions or billions of entries. We use a distributed directory service based on the Pastry distributed hash table [5], supplemented with active caching as in Beehive [6].

Grains, despite obvious similarities, are not identical to objects. The key difference is that grains cannot share memory and can only communicate by asynchronous messages. Orleans does not prescribe the size of grains. Granularity is a tradeoff between the level of parallelism and state locality needed for efficient computations. Small grains typically hold entities that are logically isolated and independent. A user account grain or a catalog item grain is a typical entity that is not data dependent on other grains of the same type. At the other end of the spectrum, a complex data structure such as a search index may be more efficient contained in a grain and accessed as a service.

2.1.1 Security

A grain can only communicate with another grain through a reference to it, which provides an object-capability-like security model [7]. Each grain in Orleans is created within a logically isolated workspace, a **grain domain**, which delimits the set of grains it can directly access. A newly created grain is initialized with explicit references to other grains and to its containing domain, which can be queried to acquire references to grains in the domain. A grain can only send messages through these grain references, or to grain references it received in a message. References to grains in other domains must be passed at initialization or in a message from a grain that already has access.

Grain domains are hierarchical and nested, and they can be used to isolate parts of an application. For example, a Orleans application may create a grain domain per customer, so code executing on behalf of a customer has limited and controlled access to grains from other customers and global application state. The application can further limit access by controlling the lookup mechanisms the customer-specific grain uses to find other grains.

2.2 Transactions

Orleans offers transactions for three reasons. The first, similar to a primary motivation for transactional memory, is to isolate concurrent computations, eliminating the need for explicit synchronization to coordinate shared data access. In Orleans, multiple activations can modify grain state, which requires a mechanism to resolve conflicts and ensure a consistent data view.

The second reason arises from the replication of data and computation, a fundamental requirement on large-scale computing on commodity hardware. Modifying the copies of a replicated datum can introduce inconsistencies. Many cloud systems trade efficiency for convenience by providing an eventual consistency data model [3, 8], which offers no guarantees beyond that an update will eventually propagate to all servers. This model complicates software development by requiring a programmer to distinguish data consistency requirements and to develop storage solutions for data that requires stronger guarantees.

Orleans instead offers a single data model based on lightweight, optimistic transactions (Sec. 4.3). Orleans transactions are isolated from other concurrently executing transactions and prevent access to data that has been modified by a transaction that has not yet completed. Transactions atomically succeed or fail and their changes become visible atomically when

the transaction completes execution. The Orleans runtime durably and atomically persists completed transactions.

The Orleans consistency model is similar to, but slightly weaker than, snapshot isolation (SI). Under SI, a transaction sees a consistent snapshot of committed updates as of the time the transaction starts. In Orleans, a transaction sees an atomically consistent subset of completed transactions. These snapshots are either the persistent stored state of a committed grain or an activation from a completed but as yet uncommitted transaction. The snapshot of a completed transaction includes all grain activations changed during the transaction and the transactions they transitively depended upon [9]. The changes made by a completed transaction become visible to future transactions atomically. However, since a transaction can become visible before committing, aborting a transaction can force aborting other, dependent transactions that have completed but not yet committed. Moreover, with multiple completed but uncommitted activations of a grain, concurrent transactions may see different combinations of atomically consistent states. This design, although weaker than serializability or snapshot isolation, maximizes responsiveness and system throughput and does not require global coordination.

The third reason for transactions is to reduce the need for explicit error handling code. Error handling is particularly difficult in an asynchronous distributed system, as many operations execute concurrently, and recovering from an error and restoring state can be very complex. Moreover, error recovery is often the least-tested and most error-prone code in an application [10]. With the Orleans transactional model, an error during a transaction causes it to abort. The runtime will either replay the operation until it succeeds or fail the transaction atomically.

2.2.1 Discussion

In large-scale systems with hundreds or thousands of servers, failure is a constant. An essential aspect of any distributed framework is its model and mechanisms for handling failures. At the level of a single user request, Orleans relies on transactions to ensure that requests are processed completely and correctly. If any participant in the transaction fails, due to hardware or software problems, the transaction rolls back and can be replayed. Optimistic transactions ensure a quick user response in the common case, while preserving consistent and correct system state in failure case.

Orleans transactions try to avoid global coordination or locking, which could adversely affect scalability, throughput, and distributed operation. An executing transaction sees a state produced by completed transactions and the transactions they transitively depend upon [9]. Orleans does not ensure full serializability, as that would require global locking or a potentially high rate of conflicting aborts. A programmer, however, can achieve stronger guarantees by constraining some grains to a single activation, at the cost of reduced scalability and fault-tolerance.

2.3 Communications

In Orleans, all communications occurs through asynchronous message exchanges between grains. Most messages follow a request/reply pattern. Grains may designate certain messages as event notifications, sent as one-way messages, and in the future we anticipating using a single-request, multiple-reply pattern for data streaming.

As with most programming models, messages are exposed as method calls. Unlike traditional RPC models, however, these calls return immediately with a **promise** (Section 3.1) for a future result, rather than blocking until the result is complete. Promises resolve the impedance mismatch between synchronous method calls and asynchronous message passing [11, 12].

2.3.1 Discussion

Promises are well suited to coordinating concurrent computations [12]. In Orleans, their primary use is to allow a grain to start one or more computations in other grains and to overlap their executions. Since computations produce results in unpredictable orders, it is convenient to associate a handler with a result, to processes it when it is produced. Unpredictability in timing introduces non-determinacy into a computation, but in a limited form that is clearly delimited and easily understood. When even this non-determinacy is problematic, promises can also be used like futures, with a blocking operation to retrieve their result.

2.4 Scalability and Resource Management

A service is scalable if increased load does not degrade its quality. Constructing a scalable service is a challenge. It must be designed and constructed to avoid bottlenecks, such as a centralized resource, that cannot grow or expand to handle increased load. Scalable solutions generally are partitioned and replicated, so they can expand by adding more hardware and redistributing their load among N+1 systems.

The two primary mechanisms in Orleans that support scalability are the grain model itself and grain activations. The grain model encourages partitioning of state and services into fine-grained, replicatable services. Those fine-grained services operate concurrently and their progress depends only on the other grains they explicitly communicate with. There are no system-wide bottlenecks that limit scalability. The programmer still must partition a service into units with appropriate granularity. In addition, Orleans improves performance by automatically growing or shrinking the number of activations depending on the demand on a grain. Activations can be placed on different servers and migrated between servers to balance the load. Finally, grains efficiently support a finer granularity of resource management than other distributed frameworks, particularly service-oriented architectures in which the unit of granularity is a process or a virtual machine. Small grains offer the Orleans runtime more flexibility in adaptively responding to changes in load by acquiring (or releasing) servers.

The initial version of the Orleans system uses a simple load-balancing policy in which requests are randomly distributed to running activations. We are currently experimenting with different resource allocation policies. In particular, we are incorporating server load in distributing requests and placing new activations, as well as the data location (similar to [13]). Effectively, our runtime will dynamically decide between transferring functions and data, based on a cost model. We also plan to make use of the network topology and the failure domain structure to minimize the latency of each transaction while ensuring the availability of the application and maximizing the overall throughput.

2.4.1 Discussion

A design goal for grains was to provide the Orleans system with the mechanisms sufficient to achieve a high degree of automatic scalability. A grain is a homogeneous computation, identifiable by its type, allowing the system to use the past

behavior of grains of a particular type to inform future performance tuning decisions. Grains also encapsulate the minimum application state necessary to perform a computation, which makes them less costly to move between servers than other encapsulation mechanisms, such as virtual machines.

2.5 Maintainability and Versioning

Rapid software evolution is a key differentiator of cloud services, as compared to conventional software, and consequently cloud applications are in a constant state of flux. Components are developed by different teams and updated on independent schedules. New components are typically introduced gradually, starting with deployment to a single server and evolving to being the production version. Unfortunately, emergency deployments sometimes must fix critical bugs and it can be necessary to roll back a deployed component. All this results in an environment where different versions of component run concurrently and components interact with multiple versions of other components.

Supporting this mix-and-match environment is a major area for future work in Orleans. Some key elements are:

- Management controls that allow programmers to control the introduction of new versions of grain classes into a running system.
- A flexible serialization format that allows interface evolution. To the extent possible, adding an optional parameter to or removing an inessential parameter from a method signature should not cause requests from older clients to fail.
- A flexible persistence format that allows forward and backward schema evolution; i.e., the state saved by a newer grain version may need to be restored to an older version, or vice versa.

3. Programming Model

This section describes key aspects of Orleans programming model, as seen by a programmer. A client is code that consumes services of a grain – either another grain, or non-grain code that interoperates with a Orleans system.

3.1 Asynchrony and Promises

Orleans uses promises as its asynchrony primitives. An instance of these types represents a promise for future completion (`AsyncCompletion`) or for a future result value (`AsyncValue<T>`) from an operation. Promises are fully integrated with the .NET async models.

```
AsyncValue<int> intPromise = GetA();
try {
   // synchronously waits for resolution of promise
   int a = intPromise.GetValue();
}
catch(Exception exc) {
   // if promise is broken, GetValue throws exception
   Console.WriteLine("Error: " + exc.Message);
}
```

Figure 1. Synchronous wait.

Promises have a simple lifecycle. Initially, a promise is *unresolved* – it represents the expectation of receiving a result at some unspecified future time. When the result is received, the promise becomes *fulfilled* and the result becomes the value of the promise. If an error occurs, either in the calculation of the result or in the communications between the requesting and responding grains, the promise becomes *broken* and has no value. A promise that has been fulfilled or broken is considered *resolved*.

```
AsyncValue<int> intPromise = GetA();
//anonymous method runs when promise is resolved
intPromise.ContinueWith((int a) => {
   // success block
   Console.WriteLine("Result: " + a.ToString());
},
(Exception exc) => {
   // exception block
   Console.WriteLine("Error: " + exc.Message);
}).Ignore();
```

Figure 2. Asynchronous continuation.

A caller that invokes an asynchronous operation immediately receives a promise and can do one or more of the following:

- Synchronously wait for the promise to resolve with the `Wait()` or `GetValue()` method. Both methods optionally take a timeout.
- Schedule a **continuation action** with the `ContinueWith()` method. It will be run when the promise becomes resolved. `ContinueWith()` takes two function delegates: one for the case when the promise is fulfilled (success closure) and the other for when it is broken (optional exception closure).
- Return the promise, if the caller itself is an asynchronous method with a compatible return type.
- Join the promise with other promises, producing another promise that will be resolved when all the joined promises resolve, and broken if any of the joined promises are broken.
- Explicitly ignore the outcome of promise resolution with the `Ignore()` method. This instructs the runtime not to propagate an error, should the operation fail. Otherwise, the caller must handle the promise with one of the other methods.

If a promise with a scheduled continuation is broken, the success closure will not be called and the exception closure, if provided, will be. The exception closure has a chance to recover from the error condition or re-throw the exception. An exception closure is analogous to a catch block for asynchronous structured exception handling.

Promises make asynchrony explicit in code, so a programmer can directly express the desired interleaving and pipelining of operations and can avoid false assumptions about concurrency. Except for the synchronous `Wait()` and `GetValue()` methods, which should be used rarely, the other alternatives do not block a thread while an asynchronous operation computes. This leads to better parallelism and more efficient usage of resources.

An activation's computation and closures execute in a **turn**-based model (Section 4.1), that executes at most one thread at a time in a cooperative multitasking manner.

3.2 Grain Interfaces

A grain class implements one or more public grain interfaces that define the grain's service contracts. A grain interface is an interface that adheres to the following rules:

1. A grain interface must directly or indirectly inherit from the `IGrain` marker interface.

2. All methods and property getters must be asynchronous, i.e. return `AsyncCompletion` or `AsyncValue`. This explicitly exposes the asynchronous nature of grain calls, makes client and server use the same interface, and allows a grain to return an unresolved promise instead of a concrete value.
3. No property setters and no subscribe/unsubscribe to events (they are synchronous in .NET).
4. Method arguments must be grain interface types or serializable types that can be logically passed by value.

For example, a simple grain interface is:

```
public interface ISimpleGrain : IGrain {
  AsyncValue<int> A { get; }
  AsyncCompletion SetA(int a);
  AsyncCompletion SetB(int a);
  AsyncValue<int> GetAxB();
}
```

3.3 Grain References

A grain reference is a proxy object that provides access to a grain. It implements the same grain interfaces as the underlying grain. A grain reference is the only way that a client, whether another grain or a non-grain client, can access a grain. Grain references can be passed as arguments to a grain method. Grain references share some similarities with promises:

- A grain reference can be in one of the three possible states: unresolved, fulfilled or broken.
- A caller can schedule and pipeline operations on a grain reference before it is resolved, by invoking an asynchronous method or using the `ContinueWith()` method. A grain reference also supports the `Wait()` method that synchronously waits for its resolution.
- Error conditions are propagated to continuation actions (by breaking promises associated with the operations) or can be handled synchronously though a `Wait()` call.

3.4 Creating and Using Grains

For each grain interface, the Orleans ClientGenerator tool generates a public factory class and an internal proxy class that convert method calls into messages. Clients use type-specific factory classes to create, find, or delete grains. In the simplest case, a factory includes methods for creating and deleting a grain and for an asynchronous cast operation. Optional annotations on grain interface members, such as `Lookup` and `Queryable`, cause ClientGenerator to add additional methods to the factory class for looking up a grain or for searching for grains that satisfy specified conditions within a given grain domain. The generated factory class for the sample interface above looks like the following:

```
public class SimpleGrainFactory {
   public static ISimpleGrain CreateGrain();
   public static void Delete(ISimpleGrain grain);
   public static ISimpleGrain Cast(IGrain grainRef);
}
```

Below is an example of the code to create a `SimpleGrain` and perform operations on it:

```
ISimpleGrain grain = SimpleGrainFactory.CreateGrain();
AsyncCompletion setAPromise = grain.SetA(3);
AsyncCompletion setBPromise = grain.SetB(4);

// join the promises
AsyncCompletion setPromise =
  AsyncCompletion.Join(setAPromise, setBPromise);
AsyncValue<int> getPromise = setPromise.ContinueWith(
    () => {
          return grain.GetAxB();
    });

// schedule action when GetAxB returns actual result
AsyncCompletion resultPromise =
   getPromise.ContinueWith((int x) => {
        Console.WriteLine("Result: " + x.ToString());
     },
     (Exception exc) => {
        Console.WriteLine("Error: " + exc.Message);
        throw exc; // re-throw the exception
     });
// wait for operation to complete
try {
    resultPromise.Wait();
} catch(Exception exc) {
   // error at any stage will throw exception
   Console.WriteLine("Error: " + exc.Message);
}
```

`CreateGrain` immediately returns a reference to the grain, allowing pipelining of asynchronous requests to the grain, such as `SetA` and `SetB`, even before the grain is fully created. The client invokes `GetAxB` on the reference before `SetA` and `SetB` fulfill their promises. The invocation is queued on the grain and executes after `SetA` and `SetB` execute. When the `getPromise` is resolved, a success or an error function delegate is invoked.

Because every asynchronous operation, such as a call to a grain method, a call to `ContinueWith()` on a promise, or a call to `Join()`, returns a promise, and because promises propagate errors through continuations, error handling can be implemented in a simple manner. A client can build an entire dataflow graph of interconnected asynchronous continuations and defer error handling until a later point. In the example above, an error at any stage of the program (`CreateGrain()`, `SetA()`, `SetB()`, `GetAxB()`, `x.ToString()`, etc.) will eventually break `resultPromise` and cause `resultPromise.Wait()` to throw an exception with information about the error to the one error handling statement (`try`/`catch`) as the top level. All possible errors bubble up to that point in the program, even though pieces of the computation may run concurrently on different threads. This greatly simplifies the error handling code.

3.5 Grain Classes

As already mentioned above, a grain class implements one or more grain interfaces. Since each method or property within a grain interface is asynchronous – returns a promise (`AsyncCompletion` or `AsyncValue<T>`) – the corresponding implementation method of the grain class has to return a promise. There are two possible cases: the method can either return a concrete value (which gets automatically converted into a resolved promise by the runtime) or return a promise that it obtains from calling another grain or scheduling a closure. For example, the `GetAxB()` method of `ISimpleGrain` can return a concrete integer:

```
AsyncValue<int> GetAxB() {
   int x = this.a * this.b;
   return x;
}
```

or return a promise obtained from another method:

```
AsyncValue<int> GetAxB() {
   AsyncValue<int> p = anotherGrain.GetAxB();
```

```
    return p;
}
```

3.6 Grain State

The state of a grain is managed by the Orleans runtime, including initialization, persistence, replication, and migration. A grain exposes its internal state to the system, and to the programmer, through virtual properties. The programmer optionally annotates the properties to declaratively specify requirements such as persistence, initialization only, etc. Because these properties are virtual, Orleans can override them by subclassing the grain class and intercept reads and writes. In the future, we plan to use this mechanism to implement caching, replication, synchronization, and other state management.

4. Implementation

Orleans is a library written in C# that runs on the Microsoft .NET Framework 4.0.

4.1 Reentrancy, Interleaving and Scheduling

Orleans is built on a cooperative multitasking model. A grain activation operates in discrete units of work called **turns** and finishes each unit before moving on to the next. A turn executes the computation to handle requests from other grains or external clients and to run closures at the resolution of a promise.

While a system may execute many turns belonging to different activations in parallel, each activation always executes its turns sequentially. Therefore, execution in an activation is always logically single threaded. This is achieved without dedicating a thread to an activation or request. Instead, the system uses a scheduler that fairly multiplexes turns on a pool of threads.

To a large extent, there is no need for locks or other synchronization methods to guard against data races and other multithreading hazards. However, promises are resolved asynchronously, so the order in which closures for different promises execute is unpredictable. This interleaving never results in a fine-grained data race, but it does require attention since the state of the activation when a closure executes may differ from its state when the closure was scheduled. In addition, if code blocks on a promise using the `Wait` method, the current turn ends and a new one starts. When the promise is resolved, the blocked code resumes in a new turn. Other turns in the same activation may have executed while `Wait` is suspended, and thus the state of the activation (including the values of properties and fields) may change between the invocation of the `Wait` method and its return. This is also true for continuations bound to a promise, but using `Wait` is more error-prone, since the state changes appear to occur in the middle of straight-line code. For this reason (among others), the use of `Wait` is discouraged.

By default, Orleans requires an activation to completely finish processing one request before accepting the next request. An activation will not accept a new request until promises created (directly or indirectly) in the processing of the current request have been fulfilled and all associated closures are executed. Grain implementation classes may be marked with the `Reentrant` attribute to indicate that turns belonging to different requests can be freely interleaved. Methods that are marked `ReadOnly` are assumed to also be reentrant.

4.2 State

To build reliable and scalable cloud applications, the application's state must be stored on persistent media and replicated in multiple physical locations. Data stored on persistent media such as hard disks can survive server crashes and power outages, but is unavailable when its host server is not operating or is disconnected from the network. Replicating data on multiple servers in physically distinct locations increases data availability and reliability, as several independent failures must occur before data is unreachable or unrecoverable. Furthermore, data replication allows an application to service requests on different servers, increasing access bandwidth and decreasing latency.

Persistence and replication change the semantics of application code and hence cannot be applied automatically or hidden from a programmer. Persistent data can be visible and shared by multiple computations, so it is necessary to clearly define the lifetime and visibility of this type of data and provide concurrency control to ensure that it is updated consistently. Replication also introduces challenges in ensuring that updates to distinct copies propagate in a predictable manner. These aspects of data semantics could be exposed by defining new classes, but in Orleans, we use annotations on data type declarations to specify persistence, replication, and consistency. Annotations offer a more expressive language for these non-trivial specifications, while cleanly separating application requirements from the complex algorithms for implementing persistence and replication.

4.2.1 Persistence

An individual grain declares the parts of its state that are persistent ("hard") and transient ("soft") through optional annotations. It can also declare methods to be called on grain activation to initialize transient state and on deactivation. Persistent property types must support serialization, including data, grain references, and resolved promises. Transient state can be stored in fields, or in properties marked with the `Transient` attribute, and can be of any type. An activation will only be activated or deactivated outside of a transaction, so its per-request state is typically transient.

At the level of a single grain, these declarations provide a simple model for persistence. The Orleans runtime activates a grain with its persistent properties initialized, either from grain creation parameters or from the latest version in the persistent storage. The grain's `Activate()` method is then called to allow it to initialize its transient state. The runtime then invokes methods to handle requests sent to the activation, which can operate freely upon their entire state in memory. To commit an activation, the runtime waits for the completion of a request, calls the grain's `Deactivate()` method, and writes the grain's state property values to persistent storage. The frequency of committing values to storage depends on the resource management policy, trading efficiency of combining multiple writes against the risk of replaying more transactions in the event of failure. Furthermore, the runtime coordinates commit operations across multiple grains to ensure that only consistent state is committed (Section 4.3.4).

4.2.2 Migration and Replication

Given a persistence mechanism, it is straightforward to migrate a grain from one server to another. Once an activation finishes processing its current request, the runtime calls the `Deactivate()` method, serializes persistent state, and destroys the old activation. The runtime then creates a new activation at the target server, initializes its state by deserializing the data, and calls the `Activate()` method. Orleans also updates its distributed directory by unregistering the old activation and registering the new one (with a new ID – activation IDs are never reused).

Requests on the destroyed activation will trigger a new directory lookup and message redirection to the new activation.

Replicating a grain is a similar process, except that the old activation is not destroyed. If existing activations are busy, the runtime can create the replicated activation directly from persistent storage instead of cloning another activation.

4.3 Transactions

Multiple activations of a grain, while beneficial for performance and fault-tolerance, raises the difficult problem of distributed consistency. Orleans's solution is a lightweight, optimistic transaction model. As a transaction executes, it communicates with activations, which join the transaction and remain joined until the transaction commits or aborts. At most one activation of a given grain may join a transaction, to ensure that computations in the transaction see a consistent view of the grain. An activation may join at most one write transaction, or any number of read-only transactions, ensuring that the transactions are isolated from other one another. When a request from a client is completely processed, a response is optimistically returned and the joined activations are released. An activation may join a series of transactions before its state is committed to persistent storage.

However, if a subsequent transaction modifies one of these activations before writing its state to persistent storage, and the transaction aborts, then the earlier transactions also must abort and re-execute since their state was not persisted. Re-execution is non-deterministic and may produce a different result. If this possibility is semantically significant, a client may opt to wait until a transaction fully commits before consuming its response. In most cases, however, the earlier and re-executed transactions are semantically equivalent, and the client need not wait until the service's state is fully committed.

4.3.1 Isolation

To ensure transaction isolation, Orleans ensures a one-to-one mapping between activations of a grain and transactions. Each server records the transaction to which each activation on the server is joined. Every message between activations contains a transaction header: a transaction identifier and list of activations already joined to the transaction. The list records incremental updates, reducing the header to the transaction identifier in the common case that both activations are on the same server.

If a message arrives for an activation already joined to a different transaction, it will be either enqueued until the activation is released from its current transaction or redirected to a different activation (perhaps newly created). The resource management policy makes a choice among these alternatives, trading the expense of redirecting a message or creating a new activation against the wait for completion of the active transaction.

Furthermore, the Orleans runtime takes advantage of the activation/transaction mapping when sending messages. If the target grain already has an activation joined to the active transaction, the message is sent directly to that activation. Moreover, the sending server can avoid activations of the target grain that are known to belong to other transactions.

4.3.2 Consistency

Joining activations to transactions also ensures a consistent view of state since there is only a single instance of an activation in a transaction. Maintaining this property is easy for applications that execute sequentially across a series of grains. Each request or response message contains the entire set of activations joined to the transaction so far. This mapping ensures that subsequent messages to the grain go to the same activation. When the application issues concurrent requests, an additional mechanism, described below, is required.

In Figure 3, activation 1a (activation "a" of grain "1") sends concurrent messages to 2a and 4a, both of which concurrently send messages to grain 3. The Orleans runtime tries to ensure that 2a and 4a send to the same grain activation, without using any distributed coordination mechanism which would be expensive and non-scalable. Both grains use the hash of the transaction identifier to index the list of available activations. If this heuristic fails and they choose different activations, say 3a and 3b, the inconsistency will be discovered when the responses arrive at 1a. The transaction will be aborted and replayed, before any code can observe inconsistencies between the state of 3a and 3b. The replay is notified of the cause of the failure, and the runtime selects one activation of grain 3 to join to the transaction before restarting grain 1. This precludes encountering the same inconsistency by ensuring that grains 2 and 4 will choose the same activation of grain 3.

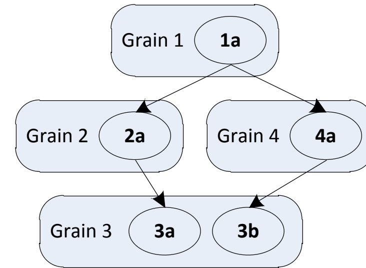

Figure 3. Consistency failure if the transaction sees two different activations (3a and 3b) of a grain.

4.3.3 Atomicity

To preserve atomicity, Orleans must ensure that the updates that a transaction made to its set of grains are either committed together or discarded. The runtime keeps the transaction/activation mapping information until transactions are committed (Section 4.3.4). Before joining an activation to a transaction, it verifies that this action preserves atomicity: for all grains that the active transaction shares with a prior, uncommitted transaction, it also shares the same activations.

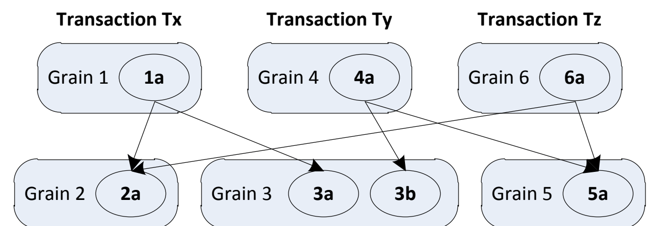

Figure 4. Potential atomicity violation. Transaction Tz cannot use either grain 3a or 3b without violating the consistency of Tx or Ty, respectively.

For example in Figure 4, completed transaction Tx has modified activations 1a, 2a, and 3a, and completed transaction Ty has modified 4a, 3b, and 5a. Active transaction Tz has modified activations 6a and 2a and sends a request to grain 5. If this

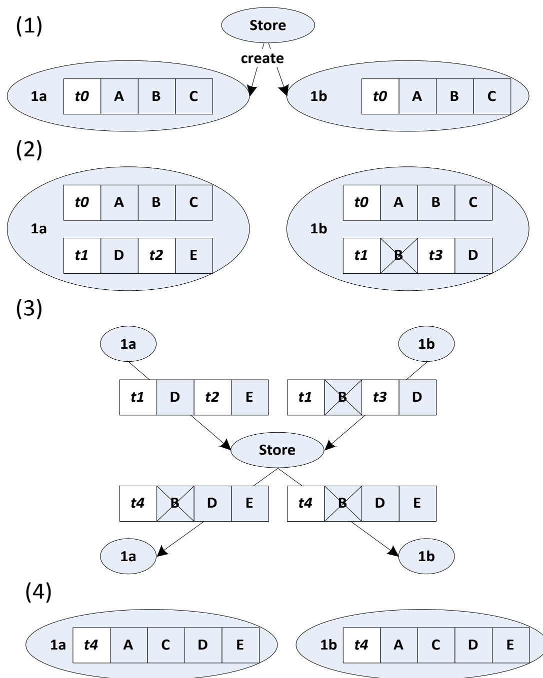

Figure 5. Reconciliation of conflicting changes in two grains.

message arrives at activation 5a, the runtime has enough information to detect a potential – but not yet actual – violation of atomicity if Tz were to send a message to grain 3. It might choose to redirect the message to another activation of grain 5. Or, if none is available and it is too expensive to create a new one, it may go ahead and join activation 5a to Tz. So far, atomicity is preserved. However, if Tz does send a message to grain 3, the runtime cannot choose either activation 3a or 3b without violating atomicity (of Ty or Tx, respectively). The runtime will detect this before the message to grain 3 can be sent and abort Tz, ensuring that no executing code observes an atomicity violation. Transactions Tx and Ty will also need to abort and replay because their updates to 2a and 5a will be lost when Tz aborts.

The set of transactions and activations linked by atomicity constraints can become large, and traversing the history graph can be time consuming. Orleans uses a more efficient data structure that summarizes dependency relationships and potential atomicity conflicts and merges dependency sets when an activation is added to a transaction.

4.3.4 Durability

Orleans needs to ensure that committed transactions are written atomically to persistent storage. The transaction persistence mechanism also follows an optimistic path, asynchronously writing modified results to storage without delaying an executing transaction. When a transaction completes, the server that handled the initial request sends a completion notification to the system store, listing all activations involved in the transaction. Committing a transaction has three phases:

1. The store collects serialized representations of the persistent state of each activation in the transaction.
2. If more than one version of a grain is to be committed, either from a more recent version already committed to the store, or from multiple transactions, their state must be reconciled to produce a single version before writing it to store.
3. The grain states are written to persistent storage using two-phase commit to ensure that all updates become visible simultaneously.

This process runs without blocking executing transactions and so can step back to an earlier phase as additional notifications arrive. The system store is currently a single point of failure and a scalability bottleneck. In the future, a distributed persistence mechanism will remove these limitations.

4.3.5 Reconciliation

If the persistent state in multiple activations of a grain is concurrently modified by different transactions, the changes must be reconciled into a single, consistent state before the transactions can commit. To handle common cases, the Orleans runtime provides synchronizable data structures that track fine-grained updates and reconcile conflicting changes. The reconcilable data structures include records, lists, and dictionaries. If an application requires a different mechanism or more complex data structures, a grain can provide a custom synchronization method to reconcile changes from two activations.

Reconciliation occurs as application state is written to persistent storage, as described above. The reconciliation model is a simple star topology, in which each modified activation is successively reconciled with the latest persistent version, and the combined version is sent back to individual servers to update each modified activation. Figure 5 illustrates the reconciliation process:

1. Two activations 1a and 1b of grain 1 are initially created from persistent storage at time t_0 with identical states holding the set {A, B, C}.
2. They participate in separate transactions. Activation 1a adds {D, E} to its set, while 1b removes B and adds D. Each change has a timestamp that can be used to resolve conflicting changes.
3. When these transactions complete, the activations send their deltas for reconciliation. The store reconciles them into a single consistent set of changes, writes the reconciled result to persistent storage, and sends the resulting changes back to the activations.
4. The store also includes a new baseline timestamp t_4, so the activations can consolidate previous changes in their baseline set and discard individual timestamps.

5. Applications

We will describe two applications built on Orleans to illustrate the flexibility of its architecture and programming model.

5.1 Instant Messaging

Instant messaging (IM) is built on small grains. Each user has an account grain, holding the user's name and email address, as well as an address book with a set of contacts. The account grain exposes a public interface IContact with information about user's presence and an owner-only IAccount interface with administrative operations to change a password, add/remove contacts, change presence status (online/offline), as well as an operation to start a conversation. Each conversation is managed by a conversation grain. Once created, a conversation grain allows its participants to add or remove participants and post and receive messages. A system administration grain exposes an interface to

create user accounts and authenticate a user (omitted, due to space). Observers enable clients to receive asynchronous notification about friends' presence changes, invitations to join a conversation, or newly posted messages. Figure 6 contains partial interfaces for IM.

IM naturally partitions by user and conversation into many independent units of functionality and state, which map well to the grain model and help the system scale. The code does not depend on a large data structure, such as a table of all user accounts, which would be complex and expensive to partition and replicate as a single entity. Instead, the programmer provides the application logic, and Orleans replicates account and user grains, provides a distributed directory to find these grains, and manages persistence and dynamically balances load by migrating grains.

```
public interface IContact : IGrain {
   [Queryable][Lookup][InitOnly]
   AsyncValue<string> Username { get; }
   AsyncValue<Presence> Presence { get; }
   AsyncCompletion AddConversation(IConversation c);
   AsyncCompletion RemoveConversation(IConversation c);
}
public interface IAccount : IGrain, IContact {
   AsyncValue<List<IContact>> AddressBook { get; }
   AsyncCompletion AddContact(IContact contact);
   AsyncCompletion RemoveContact(IContact contact);
   AsyncCompletion SetPresence(Presence presence);
   AsyncValue<bool> ValidatePassword(string password);
   //methods to add and remove conversation observers
}
public interface IConversation : IGrain {
   AsyncCompletion AddParticipant(IContact contact);
   AsyncCompletion RemoveParticipant(IContact contact);
   AsyncValue<List<IContact>> Participants { get; }
   AsyncCompletion AddMessage(InstantMessage message);
}
```

Figure 6. Instant messenger grain interfaces.

5.2 Large Graph Processing Engine

A very different kind of Orleans application is our library for processing large distributed graphs. Graphs are central to social networking and other web applications. Large graphs pose many challenges, as they do not fit a single computer and distributed algorithms are communications intensive [14]. Our graph library provides support for partitioning and distributing graph data (nodes, edges, and metadata) across machines and for querying graphs.

Orleans offers two options for implementing graphs: encapsulate each node in a separate grain or have a set of nodes (called a partition) represented by a grain. We picked the latter approach to reduce memory overheads and increase the size of the graph we can handle, and to reduce communication cost in this communication-intensive application. Every server hosts a small number of partition grains, and every partition grain contains a large number of nodes ($10^4 – 10^6$). A graph algorithm running in a partition directly accesses nodes in its partition. Accesses across partitions involve messages, which become more expensive when they cross machine boundaries. The graph algorithms are aware of this distinction and batch message between servers to reduce communication overhead.

Graphs demonstrate the flexibility of Orleans model: it imposes no restrictions on the size of a grain. Grains can hold potentially large amounts of state, while still offering isolation, asynchronous messaging, persistence, and transactional updates. The graph library is built upon an abstract graph execution framework that offers functionality similar to Pregel [14], and similar abstract frameworks could be built in Orleans to support high-level patterns such as Map/Reduce [13], and Dryad [15].

6. Performance Measurements

We analyze the performance of Orleans through a set of benchmarks. The measurements are averages over 1000 runs on a cluster of Intel Core 2 Duo CPUs at 3.16GHz, 4GB of RAM, and 64 bit Windows 7.

Figure 7 depicts the round trip latency of a grain method invocation, for garins locatd on the same and different servers. The method invocation had one paramter, a byte buffer of varying size. The latency for the remote case is around 0.7 millisecond and half that for the local case. For large messages the latency increases proportionaly to the message size, due to the cost of serialization.

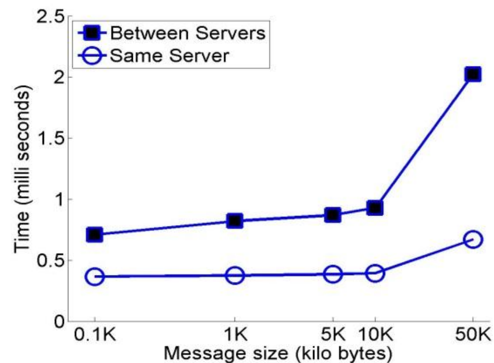

Figure 7. Local and remote invocation latency.

We also measured the overhead of promises. The time to create a promise and join its result in continuation is between 50 to 100 microseconds. This is small compared to the message latency.

A latency to create a new grain is approximately 5 milliseconds, which includes creation of a first activation for this grain and registering it in a distributed directory.

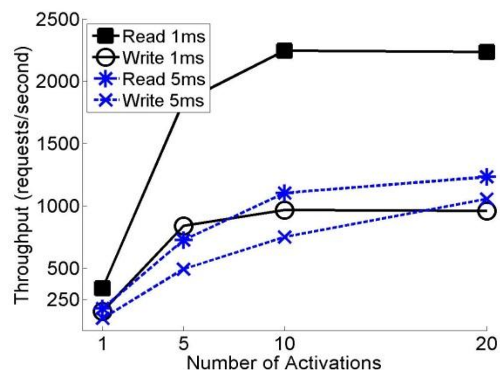

Figure 8. Throughput of multiple activations.

Figure 8 shows the throughput of requests to a grain as the runtime automatically increases the number of activations to distribute the load. Each request is either read or write, performing 1 ms or 5 ms of work, respectively. The write transactions impose

additional system load for the commit and reconciliation protocols. While the increase in throughput is sub-linear (a bottleneck we are investigating), this illustrates a main feature of Orleans. Significant increases in load can be handled transparently by adding more activations, without application intervention.

7. Related Work

Orleans borrows many concepts from prior systems. However, we believe that combination of design choices is unique and tailored to provide a comprehensive solution to the new domain of scalable client + cloud applications.

7.1 Actor Models

Actors are a well-known model for concurrent programming that form the basis for many programming languages [16] including: Erlang [17], Ptolemy [18], and E [18].

Orleans differs from classic actors in Orleans's mutability of grain state, its asynchronous communication and promises, its message ordering guarantees, its support for replication, and its transactional model and consistency guarantees.

Orleans differs fundamentally from Erlang in its imperative language semantics and in its rich, distributed runtime support. Erlang, unlike Orleans, does not provide data migration. Erlang messaging is exposed via synchronous RPC or direct messaging, while Orleans provides remote method invocation with higher-level primitives (promises). Erlang also supports transactions and failure replication, through libraries, though with less-scalable semantics than Orleans. Erlang also differs from Orleans in its distributed error handling mechanism, which requires a programmer to implement guard processes to handle system failures, while in Orleans system failures are handled automatically by the transaction system.

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems, with the focus on assembly of concurrent components. Ptolemy does not provide a distributed runtime.

E is an object-oriented programming language for secure distributed computing. E has a concurrency model similar to Orleans, based on event loops and promises, but its unit of isolation and distribution is much larger: a "vat" containing many objects that can share state. E also lacks Orleans's distributed runtime support for persistence, replication, migration, and transactions.

7.2 Distributed Object Models

Enterprise Java Beans (EJB), Microsoft's Component Object Model (COM), and the Common Object Request Broker Architecture (CORBA) are all object-oriented frameworks for building three-tiered applications. While they differ in detail, all are based on distributed objects, (primarily) synchronous RPCs, location transparency, declarative transaction processing, and integrated security. They share Orleans's goals of offering a higher-level collection of abstractions that hide some of the complexity of building distributed systems, but are targeted at enterprise rather than cloud-scale applications.

Orleans differs from these in its embrace of strongly-typed asynchronous APIs as the programming model for all application component access. Orleans's use of multiple activations of individual grains executing concurrently and replicated across multiple nodes for scalability and failure tolerance is a significant semantic and capability difference. Its approach to consistency and transactions also makes a different trade-off between consistency and scale than the strict ACID semantics offered by the other frameworks.

8. Conclusions

We have described the design and implementation of Orleans, a software framework for cloud computing. Orleans defines an actor-like model of isolated grains that communicate through asynchronous messages and manage asynchronous computations with promises. The isolated state and constrained execution model of grains allows the Orleans runtime to persist, migrate, replicate, and reconcile grain state without programmer intervention. Orleans also provides lightweight, optimistic, distributed transactions that provide predictable consistency and failure handling for distributed operations across multiple grains. We believe that this framework will significantly simplify the development of complex cloud applications, by incorporating fundamental distributed computing functionality and abstractions into the system and by promoting the use of design patterns that promote scalability and reliability.

9. References

1. Weiser, M., *Some Computer Science Issues in Ubiquitous Computing*, in *Communications of the ACM*. 1993, ACM. p. 74-84.
2. Chang, F., et al., *Bigtable: A Distributed Storage System for Structured Data.* ACM Transactions on Computer Systems, 2008. **26**(2): p. 1-26.
3. DeCandia, G., et al., *Dynamo: Amazon's Highly Available Key-value Store*, in *21st ACM SIGOPS Symposium on Operating Systems Principles*. 2007, ACM: Stevenson, WA. p. 205-220.
4. Burckhardt, S., A. Baldassin, and D. Leijen, *Concurrent Programming with Revisions and Isolation Types*, in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 2010, ACM: Reno, NV. p. 691-707.
5. Rowstron, A.I.T. and P. Druschel, *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, in *IFIP/ACM International Conference on Distributed Systems Platforms*. 2001, Springer-Verlag: Heidelberg, Germany. p. 329-350.
6. Ramasubramanian, V. and E.G. Sirer, *Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays*, in *First Symposium on Networked Systems Design and Implementation*. 2004, Usenix: San Francisco, CA. p. 99-112.
7. Miller, M., *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control* 2006, Johns Hopkins: Baltimore, MD. p. 302.
8. Terry, D.B., et al., *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*, in *15th ACM Symposium on Operating Systems Principles*. 1995, ACM: Copper Mountain Resort, CO. p. 173-183.
9. Atul, A., L. Barbara, and O.N. Patrick. *Generalized Isolation Level Definitions*. in *16th International Conference on Data Engineering*. 2000. San Diego, CA: IEEE.
10. Weimer, W. and G.C. Necula, *Finding and Preventing Run-time Error Handling Mistakes*, in *19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 2004, ACM: Vancouver, BC. p. 419-431.
11. Liskov, B. and L. Shrira, *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems*, in *ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. 1988, ACM: Atlanta, GA. p. 260-267.
12. Miller, M.S., E.D. Tribble, and J. Shapiro, *Concurrency Among Strangers: Programming in E as Plan Coordination*, in *International Symposium on Trustworthy Global Computing*, R.D. Nicola and D. Sangiorgi, Editors. 2005, Springer: Edinburgh, UK. p. 195-229.
13. Dean, J. and S. Ghemawat, *MapReduce: a Flexible Data Processing Tool*, in *Communications of the ACM*. 2010, ACM. p. 72-77.

14. Malewicz, G., et al., *Pregel: A System for Large-scale Graph Processing*, in *International Conference on Management of Data*. 2010, ACM: Indianapolis, IN. p. 135-146.
15. Isard, M., et al., *Dryad: Distributed Data-parallel Programs from Sequential Building Blocks*, in *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 2007, ACM: Lisbon, Portugal. p. 59-72.
16. Karmani, R.K., A. Shali, and G. Agha, *Actor Frameworks for the JVM Platform: A Comparative Analysis* in *7th International Conference on the Principles and Practice of Programming in Java*. 2009, ACM: Calgary, Candada. p. 11-20.
17. Armstrong, J., *Erlang*. Comm. of the ACM, 2010. **53**(9): p. 68-75.
18. Eker, J., et al., *Taming Heterogeneity - the Ptolemy Approach.* Proceedings of the IEEE, 2003. **91**(1): p. 127-144.