Orleans: Cloud Computing for Everyone

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, Jorgen Thelin
Microsoft Research
{sbykov, ageller, gkliot, larus, ravip, jthelin}@microsoft.com

ABSTRACT

Cloud computing is a new computing paradigm, combining diverse client devices – PCs, smartphones, sensors, single-function, and embedded – with computation and data storage in the cloud. As with every advance in computing, programming is a fundamental challenge, as the cloud is a concurrent, distributed system running on unreliable hardware and networks.

Orleans is a software framework for building reliable, scalable, and elastic cloud applications. Its programming model encourages the use of simple concurrency patterns that are easy to understand and employ correctly. It is based on distributed actor-like components called grains, which are isolated units of state and computation that communicate through asynchronous messages. Within a grain, promises are the mechanism for managing both asynchronous messages and local task-based concurrency. Isolated state and a constrained execution model allow Orleans to persist, migrate, replicate, and reconcile grain state. In addition, Orleans provides lightweight transactions that support a consistent view of state and provide a foundation for automatic error handling and failure recovery.

We implemented several applications in Orleans, varying from a messaging-intensive social networking application to a data- and compute-intensive linear algebra computation. The programming model is a general one, as Orleans allows the communications to evolve dynamically at runtime. Orleans enables a developer to concentrate on application logic, while the Orleans runtime provides scalability, availability, and reliability.

Categories and Subject Descriptors
C.2.4 [**Comp.-Communication Networks**]: Distributed Systems - *Distributed applications*
D.1.3 [**Software**]: Programming techniques – *Concurrent programming.*
D.2.12 [**Software**]: Software Engineering – *Interoperability: Distributed objects.*

General Terms
Design, Languages, Performance, Reliability

Keywords
Cloud Computing, Distributed Actors, Programming Models.

1. INTRODUCTION

Writing software for the cloud poses some of the most difficult challenges in programming. Cloud systems are inherently parallel and distributed, running computations across many servers, possibly in multiple data centers, and communicating with diverse clients with disparate capabilities. Individual computers and communication links are commodity components, with non-negligible failure rates and complex failure modes. Moreover, cloud applications generally run as a service, gaining economies of scale and efficiency by concurrently processing many clients, but also facing the challenges of handling varying and unpredictable loads while 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, scalable, elastic, and efficient software.

1.1 Orleans

Orleans is a software framework with two primary goals: to make it possible for developers unfamiliar with distributed systems to build large-scale applications, and to ensure that these systems handle multiple orders of magnitude of growth without requiring extensive re-design or re-architecture. In order to meet these goals, we intentionally constrained the programming model to guide developers down a path of best practices leading to scalable applications. Where possible, we provide declarative mechanisms, so a developer specifies his or her desired behavior and leaves the Orleans runtime responsible to meet the specification.

The Orleans programming model is based on asynchronous, isolated, distributed actors. Actors can be automatically replicated to enhance scalability and availability. An actor's state can be persisted to shared, durable storage, where a reconciliation mechanism allows actors to lazily merge their state changes. Lightweight transactions provide a consistent system view across actors and simplify failure handling.

1.2 Grains

Actors within Orleans are called **grains** and are the basic programming unit. All code that a developer writes for Orleans runs within a grain. A system runs many grains concurrently. Grains, however, do not share memory or other transient state. They are internally single-threaded and process each request fully before handling the next one. Cloud services achieve high throughput by processing multiple, independent requests concurrently. Grains support this architecture with a single-threaded execution model that provides mechanisms such as isolation, consistency, and asynchrony to exploit concurrency among servers, while avoiding the error-prone difficulties of multithreading and synchronization.

Orleans does not prescribe the size of grains and supports both fine- and coarse-grain units of concurrency. Granularity must balance a tradeoff between the concurrency and state needed for efficient computations. Small grains typically hold entities that are logically isolated and independent. For example, a user account grain or a catalog item grain is often independent of other grains of the same type. At the other end of the spectrum, a complex data structure with many internal pointers, such as a search index, is more efficiently constructed in a single grain or a small collection of grains and accessed as a service. Orleans supports the full range of possibilities. We provide more examples for different grain usages and sizes in Section 4.

Grains interact entirely through asynchronous message passing. Orleans messages are exposed as method calls. Unlike most RPC models, which block until a result arrives, Orleans message calls return immediately with a **promise** (Section 2.1) for a future result. An application can bind code to the promise which will execute when a result arrives, or it can treat the promise like a future and explicitly wait for a result. Promises resolve the impedance mismatch between synchronous method calls and asynchronous message passing and are well suited to coordinating concurrent computations [1, 2]. In Orleans, the primary use of promises is to allow a grain to start one or more operations in other grains and to schedule a handler to execute when the operations complete. Unpredictable timing introduces non-deterministic interleavings among handlers, but it is limited to promises, where it is clearly delimited and easily understood.

1.3 Activations

In order to provide higher system throughput to handle increased load, Orleans automatically creates multiple instantiations, called **activations,** of a busy grain to handle its simultaneous requests. The activations process independent requests for the grain, possibly across multiple servers. This increases throughput and reduces queuing latency at "hot" grains, thus improving system scalability. If necessary, Orleans can obtain additional machines from the underlying elastic cloud service to run the new activations.

Grains are logical programming abstractions and activations are run-time execution units. For the most part, a developer can assume a single logical entity in the system processes all messages sent to a grain. Each activation of the grain runs independently of and in isolation from the other activations. These activations cannot share memory or invoke each other's methods. Their only interaction, where the behind-the-scene parallelism is exposed, is reconciling changes to the grain persistent state shared by the activations (Section 3.4).

1.4 Persistence and Reconciliation

A cloud system has persistent state that is kept in durable storage. Orleans integrates persistence into grains, which are containers of application state. A grain's state is persistent by default, which means that state changes in one activation of a grain will be available for subsequent activations of the grain (Section 3.2). In addition to persistent state, a grain may also have in-memory transient state that is not saved to durable storage and exists only during the lifetime of the activation.

A grain may exist only in durable storage – i.e., no activations on any server – when no requests for the grain are pending. In this case, when a request arrives, the Orleans runtime selects a server and creates an activation for the grain. The new activation is initialized with the grain's persistent state.

If the grain modifies its persistent state, Orleans updates persistent storage with the in-memory updates at the completion of the application request. In practice, this write may be delayed to improve performance, at the cost of increasing the window of vulnerability to failure.

Since multiple activations of a grain can concurrently run and modify the grain's state, Orleans provides a mechanism to reconcile conflicting changes. It uses a multi-master, branch-and-merge data update model, similar to Burckhard's revision-based model [3] (Section 3.3).

1.5 Consistency and Failure Handling

To simplify development and reasoning about a concurrent system, Orleans provides a simple consistency model for grains. Orleans's model is based on lightweight, optimistic transactions (Section 3.4). By default, all of the grains that process an external request are included within a single transaction, so that the request executes atomically and in isolation. Activations running in a transaction are isolated from activations running in other transactions and cannot access data modified by a transaction that has not yet completed. Transactions atomically succeed or fail and their state changes are durably and atomically persisted and become visible atomically.

Transactions also offer a simple, albeit coarse, error-handling mechanism, which is particularly valuable in failure-prone distributed systems. If a grain fails or becomes unavailable, perhaps due to a network or server failure, the transaction processing a request aborts, and Orleans automatically re-executes it. This mechanism frees application developer from handling most transient hardware or software failures.

1.6 Automated Scalability

A service is scalable if increased load can be handled by proportionally increasing server capacity. Constructing a scalable service is a challenge. It must be designed and constructed to avoid bottlenecks, such as centralized resources, that throttle system throughput under increased load. Common techniques for achieving scalability are asynchrony, partitioning, and replication. Orleans incorporates these three techniques in its programing model and runtime.

The actor model (grains) is based on asynchronous communications and encourages fine-grained partitioning of state and computation. The Orleans runtime automatically replicates activations of busy grains to dynamically spread demand and balance load across the available servers.

Unfortunately, these mechanisms cannot automatically solve all scaling problems. Partitioning and replication must be designed into an application by using the appropriate Orleans mechanisms. For example, data and processing in a single grain can become large enough that simply replicating the grain does not provide sufficient throughput. An account grain with millions of connections in a social networking application would face this problem. Refactoring a large grain into a collection of manageable grains depends on the semantics of an application and so cannot be done automatically. Orleans provides mechanisms, but the developer is responsible for using them.

1.7 Contributions

This paper makes the following contributions:

- A new programming model called Orleans for building cloud software, which makes it possible for non-expert developers to build scalable, elastic, reliable, and efficient cloud services.
- Extensions to the actor model to enhance the scalability, software elasticity, and failure tolerance of systems providing cloud services.
- A consistency model for replicated actor computations that offers weaker guarantees than traditional transactions, but greater scalability.
- Mechanisms for managing performance, persistence, isolation, and consistency that provides a simple, effective programming model with minimal application code.
- Implementation of an efficient runtime that provides good performance and high availability.

The rest of the paper is organized as follows. Section 2 describes the programming model in more detail. Section 3 describes the runtime system. Section 4 describes three sample applications, and Section 5 presents performance measurements. Section 6 surveys related work, while Sections 7 and 8 discuss future work and conclusions.

2. PROGRAMMING MODEL

This section describes the Orleans programming model and provides some code examples from the sample Chirper message-based social network application (Section 4.1).

2.1 Promises

Orleans uses promises as its asynchrony primitive. 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 in the processing of the request, the promise becomes *broken* and has no value. A promise that has been fulfilled or broken is considered *resolved*.

Promises are implemented as .NET types. An instance of these types represents a promise for future completion (`AsyncCompletion`) or for a future result value (`AsyncValue<T>`) from an operation.

The primary way to use a promise is to schedule a delegate to execute when the promise is resolved. Delegates are scheduled by calling the `ContinueWith` method on a promise; `ContinueWith` returns a promise for the completion of or the value of the delegate. If the underlying promise is broken, then the scheduled delegate does not run, and the promise returned by `ContinueWith` is also broken unless the developer provides a failure delegate; this error propagation is a key feature of the Orleans programming model.

Orleans also allows a promise to be treated similarly to an explicit future [4]. Promises provide a `Wait` method that blocks until the promise is resolved. Result value promises also provide a `GetValue` method that blocks until the promise is resolved and returns the result.

Here is an example of creating a promise by invoking a method on a grain, scheduling a delegate for when the method completes, and then blocking until the promise is resolved:

```
AsyncCompletion p1 = grainA.MethodA();
AsyncCompletion p2 = p1.ContinueWith(() =>
{
    return grainB.MethodB();
});
p2.Wait();
```

A composite promise is created from multiple promises using the `Join` method. The composite promise is resolved when all of the joined promises resolve. If any of the individual promises are broken, the composite promise breaks as well.

The execution of a delegate in an activation is always single-threaded; that is, no more than one delegate will execute at a time in a grain activation (Section 2.2).

2.2 Grain Execution Model

When an activation receives a request, it processes it in discrete units of work called **turns**. All grain code execution, whether handling a message from another grain or an external client or the execution of a delegate, runs as a turn. A turn always executes to conclusion without preemption by another turn for the same activation.

While an Orleans system as a whole may execute many turns belonging to different activations in parallel, each activation always executes its turns sequentially. Hence, execution in an activation is logically single threaded. Threads are not dedicated to an activation or request; instead, the system uses a scheduler that multiplexes turns from many activations across a pool of threads.

This single-threaded execution model removes the need for locks or other synchronization to guard against data races and other multithreading hazards. This model, however, limits parallel execution to collections of grains and hence excludes shared-memory parallelism. The restriction on parallelism within a grain was made to significantly simplify code development and avoid the host of subtle and complex errors commonly associated with shared-memory parallelism.

Orleans does not eliminate execution non-determinism. Promises are resolved asynchronously and the order in which continuation delegates 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 delegate executes may differ from its state when the delegate was created.

By default, Orleans requires an activation to completely finish processing one external request before accepting the next one. An activation will not accept a new request until all promises created (directly or indirectly) in the processing of the current request have been resolved and all associated delegates have executed. If necessary, such as to handle call cycles, grain implementation classes marked with the `Reentrant` attribute allow turns belonging to different requests to freely interleave. Methods marked `ReadOnly` are assumed to also be reentrant.

2.3 Error Handling

Because every asynchronous operation, such as a call to a grain method or a call to `ContinueWith`, returns a promise, and because promises propagate errors, error handling can be implemented in a simple manner. A client can build a complex dataflow graph of interconnected asynchronous computations and defer error handling until the result is actually needed. In the code above, an error at any stage of the program (in `MethodA` or

MethodB) will eventually break promise p2 and cause p2.Wait() to throw an exception with information about the error. All possible errors bubble up to that point in the program, even though the computations may have run concurrently on different threads or machines. Using the automatic error propagation and optional failure delegates as an asynchronous try/catch mechanism greatly simplifies error handling code.

2.4 Grain Interfaces

Rather than developing a separate interface definition language, Orleans uses standard .NET interfaces to define the interface to grain's services. An interface used for this purpose must adhere to the following rules:

- A grain interface must directly or indirectly inherit from the IGrain marker interface.
- All methods and property getters must return a promise. Property setters are not allowed as .NET does not allow them to return a completion promise. Similarly, .NET events are not allowed.
- Method arguments must be grain interface types or serializable types that can be logically passed by value.

For example, Figure 4 contains the Chirper grain interfaces.

2.5 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 are first-class values that can be passed as arguments to a grain method or kept as persistent values within a grain's state.

As with promises, grain references can be in one of the three possible states: unresolved, fulfilled or broken. A caller creates a grain reference by allocating a new grain or looking up an existing grain (Section 2.6). If operations are invoked on a reference before it was resolved, the operations are queued transparently in the reference and executed in order when the reference is fulfilled (i.e., the grain is successfully created or looked up).

2.6 Creating and Using Grains

For each grain interface, Orleans generates a static factory class and an internal proxy class. Clients use the factory classes to create, find, and delete grains. The proxy classes are used by the Orleans runtime to convert method calls into messages.

In the simplest case, a factory class includes methods for creating and deleting a grain, and for casting a grain reference of one type to a reference of another type. Annotations on grain interface members, such as Queryable, are used by the developer to cause Orleans to generate additional methods on the factory class for searching for grains that satisfy specified conditions. The generated factory class for the IChirperAccount interface looks as follows:

```
public class ChirperAccountFactory
{
  public static IChirperAccount
     CreateGrain(string name);
  public static void
     Delete(IChirperAccount grain);
  public static IChirperAccount
     Cast(IGrain grainRef);
  public static IChirperAccount
     LookupUserName(String userName);
}
```

Below is an example of the code to create a ChirperAccount grain and perform an operation on it:

```
IChirperAccount alice =
  ChirperAccountFactory.CreateGrain("Alice");

AsyncCompletion aPromise = alice.FollowUser("Bob");
```

CreateGrain immediately returns a grain reference. This enables pipelining of asynchronous requests to the grain, such as FollowUser, even before the grain is fully created. The invocation is queued on the grain reference and executes after the grain creation completes. If the grain creation fails, the grain reference would be broken, which would cause aPromise to break as well.

2.7 Grain Classes

As already mentioned above, a grain class implements one or more grain interfaces. Each grain method and property getter must return a promise. As a convenience, a method can return a concrete value, which is automatically converted into a resolved promise by the runtime. For example, the GetPublishedMsgs method of IChirperAccount can return a concrete list:

```
AsyncValue<List<string>> GetPublishedMsgs()
{
  List<string> list =
      PublishedMsgs.Skip(start).Take(n).ToList();
  return list;
}
```

An implementation method may also return a promise that it creates directly or obtains from calling another grain or scheduling a delegate.

An example below demonstrates the FollowUser method implementation:

```
AsyncCompletion FollowUser (string name)
{
  IChirperPublisher user =
     ChirperPublisherFactory.LookupUserName(name);

  IChirperSubscriber me = this.AsReference();

  AsyncCompletion p = user.AddFollower(myName, me);
  return p.ContinueWith(() =>
  {
    this.Subscriptions[name] = user;
  });
}
```

Imagine that Bob wants to follow Alice. Bob will invoke a FollowUser method on his IChirperAccountGrain passing it Alice's name. The method first looks up Alice's account grain. For that it uses the ChirperAccountFactory method that allows looking up by name, since name is declared as a Queryable property. It then creates a reference to itself by casting the C# this reference to a grain reference using the factory-provided extension method AsReference and invokes the AddFollower method on Alice's account grain. The invocation for AddFollower is queued by the runtime and is dispatched only after LookupUserName completes. Adding Alice to Bob's local list of subscriptions is queued and will execute when AddFollower completes successfully.

3. ORLEANS RUNTIME

This section describes the Orleans runtime in greater detail, focusing on the key mechanisms that Orleans provides for an application.

3.1 Platforms

Orleans is a framework for the Microsoft .NET runtime that can be used from any .NET language (C#, F#, etc.). Orleans can run on desktop machines, servers running Windows Server 2008, and the Microsoft Windows Azure cloud platform. An Orleans application remains the same when run on those different platforms.

3.2 State Management

The state of a grain is managed by the Orleans runtime throughout the grain's lifecycle: initialization, replication, reconciliation, and persistence. The programmer identifies the persistent state and Orleans handles the rest. No application code is required to persist or load the grain state.

Orleans itself does not implement storage with the required capabilities of durability and high availability; rather, it relies on an external persistence provider such as Microsoft's Windows Azure Storage.

3.3 Persistence

Each grain type declares the parts of its state that are persistent, using .NET annotations. Persistent property types must support serialization and may include data, grain references, and resolved promises.

At the level of a single grain type, these declarations provide a simple model for persistence. The Orleans runtime activates a grain with its persistent properties already initialized, either from grain creation parameters or from the current version in 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 the state in memory.

To commit an activation to storage, the runtime waits for the completion of a transaction (i.e., the end of a request), calls the grain's `Deactivate` method, and writes the grain's state property values to persistent storage. For optimistic transactions, the frequency of committing values to storage depends on the resource management policy, trading efficiency of combining writes from multiple requests against the risk of needing to replay more transaction in the event of failure. Furthermore, the runtime coordinates commit operations across multiple grains to ensure that only atomically consistent state is committed (Section 3.8.4).

3.4 Replication

The single-threaded execution model limits the amount of processing that may be performed by a single activation, and thus limits the amount of load that the activation can handle. Orleans uses grain replication – multiple activations of the same grain – as its primary mechanism to achieve software elasticity and scalability. Different activations can process independent requests for the same grain in parallel, which increases the throughput and reduces the queuing latency of the grain, thus improving system scalability.

When the current set of activations of a grain is not capable of keeping up with the grain's request load, the runtime will automatically create new activations of the grain and shift a portion of the load to them. A new activation can be created either by copying the in-memory state of an existing activation or, if the activations are busy, taking the current state from persistent storage. When the load on the grain reduces, the system will reclaim idle activations, thus reducing the amount of server resources used by the grain.

Orleans will initially place new activations on the same server as the first activation of the grain. If the local server is sufficiently busy, Orleans will create new activations on other servers of the system.

Orleans tracks the location of activations in a directory, which provides more flexibility in activation placement than schemes based on consistent hashing or other computed placements. Depending on the application, this directory may grow to millions or billions of entries. To support large applications, Orleans uses a directory service based on a distributed one-hop hash table supplemented with an adaptive caching mechanism.

The policies to decide when to create a new activation and which activation to route a request to are described in Section 3.6.

3.5 Isolation

Activations of many different grains, as well as multiple activations of the same grain, may all run on the same server in the same system process. Regardless of location, all activations communicate only through asynchronous message passing and reference each other using grain references (proxy objects). Orleans relies on the standard .NET type and memory safety guarantees to ensure isolation [5]. This allows Orleans to place activations on any server, even across data centers, and migrate activations between servers, in order to balance load, increase failure tolerance, or reduce communication latency.

3.6 Resource Management

Orleans's primary software elasticity mechanism is growing and shrinking the number of activations of a grain. Activations may also be placed on different servers and migrated between servers to balance load across the system. New user requests may be routed to any existing activation. Grains, because they can encapsulate smaller units of computation, can efficiently support a finer granularity of resource management than other distributed frameworks, particularly service-oriented architectures in which a process or a virtual machine is the unit of granularity. Grains with a moderate amount of state offer the Orleans runtime considerable flexibility in responding to changes in load, by reducing the cost of starting and migrating grain activations.

Orleans automatically manages the computational resources on which it runs. When running on an elastic infrastructure such as Windows Azure, Orleans requests new server instances to handle increasing user requests, and then starts new activations on these servers. Orleans returns server instances when load decreases and they are no longer required.

The initial version of Orleans uses a simple load-balancing and load shedding policy. Requests are initially randomly distributed to existing activations. A request arriving at an overloaded server is rejected, and the sender resubmits the request to another activation on another server. Server load is a combination of the total CPU utilization and the total number of pending requests for all activations on the server. If a request arrives at an overloaded activation (with more pending requests than a predefined threshold) on a non-overloaded server, a new activation is created on the server and the request is queued to this activation. Activations that remain idle for a sufficient length of time are garbage collected. This simple, greedy policy has worked well for the scenarios we looked at so far, but Orleans exposes hooks and mechanisms to implement more sophisticated and tailored policies.

We are currently actively experimenting with more sophisticated resource allocation policies that take into account data locality as well as compute load (similar to map/reduce [6]) in a more globally coordinated fashion. 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 request while ensuring the availability of the application in the face of failures and maximizing overall throughput.

3.7 State Reconciliation

If multiple activations of a grain concurrently modify their persistent state, the changes must be reconciled into a single, consistent state. For many applications, a last-writer-wins strategy is often sufficient, but complex data structures can benefit from fine-grained reconciliation policies. To handle common cases, the Orleans runtime provides reconcilable data structures (records, lists, and dictionaries) that track updates and automatically reconcile conflicting changes. If an application requires a different reconciliation algorithm or other data structures, Orleans provides mechanisms to allow the developer to implement them.

The reconciliation mechanism is integrated into the transactions system; its implementation is described in Section 3.8.5.

3.8 Transactions

Transactions in Orleans serve three roles:

- Isolate concurrent operations from each other.
- Ensure that an operation sees a consistent application state despite grain replication and distribution.
- Reduce the need for explicit error handling and recovery logic.

Orleans transactions are atomic, consistent, isolated, and durable. During execution, a transaction sees only a single activation of each grain involved in the transaction, thus every transaction by itself sees a consistent application state. This state is isolated from changes made by concurrently executing transactions. A transaction's updates to durable storage, even if they occur in multiple grains, become visible atomically to subsequent transactions when the transaction completes; so another transaction sees the entire, consistent set of changes from a completed transaction. Updates across many grains are atomically committed to durable storage, providing a consistent mechanism for persisting the result of a computation.

The transaction system operates by tracking and controlling the flow of execution through grain activations. A transaction is created at the arrival of an initial, external request from a client outside the system. The transaction encompasses all grain activations invoked to process the request, unless the developer specified explicit transactional boundaries. A transaction is **completed** when the request processing finishes execution. It is **committed** when its changes are written to durable storage. Orleans allows the programmer to choose to see the results of a completed transaction before it has been committed. We refer to this as optimistic transactions. The programmer can also choose to wait for the transaction to commit before seeing its results. This alternative is called a pessimistic transaction; it provides stronger consistency at the cost of higher latency and reduced performance.

In case of failures, an executing or completed transaction may be aborted and re-executed before it commits. Re-execution is non-deterministic and may produce a different result. If this possibility is unacceptable for an application, a client may mark a transaction as pessimistic. In most cases, however, the prior and re-executed transactions are semantically equivalent, and the client need not wait until the application's state is fully committed.

3.8.1 Isolation

Isolation ensures a transaction does not see changes from concurrently executing transactions, and its changes are not visible to other transactions until it completes. To ensure this, Orleans maintains a one-to-one correspondence between activations of a grain and active read-write transactions. An activation will participate in no more than one active transaction, unless all of the transactions are read-only. We say that an activation *joins* a transaction when it receives a first message within this transaction. An activation remains joined to the transaction until the transaction completes.

3.8.2 Consistency

Consistency is specified both within a transaction and across transactions.

Within a transaction, consistency requires that the sequence of observed activation states must be consistent with the partial order defined by the flow of request and response messages within the transaction. Joining activations to transactions ensures that there is only a single activation of a grain in each transaction. That is, a single transaction operates on a single copy of the grain's state. This guarantees a strongly consistent view of state within a transaction.

Maintaining this property is easy for applications that execute serially across a set of grains (i.e., grain A send a message to grain B, which sends a message to grain C). Each request or response message contains the entire set of activations joined to the transaction so far. Every time a request within a transaction is made to a grain X, the runtime picks an activation for X that is already present in this transaction's joined set. If no activation was joined so far, the runtime is free to choose any activation that is not already participating in another transaction. However, when the application issues multiple, concurrent requests, an additional mechanism is required.

In Figure 1, activation A1 (activation "1" of grain "A") sends concurrent messages to B1 and D1, both of which concurrently send messages to grain C. The Orleans runtime tries to ensure that B1 and D1 send to the same grain activation without using a distributed coordination mechanism, which would be expensive and non-scalable. If this heuristic mechanism fails and the grains choose different activations, say C1 and C2, the inconsistency will be discovered when the responses arrive at A1. At that point, the transaction aborts before any code can observe inconsistencies between the state of C1 and C2. When the transaction is replayed, it is notified of the cause of the failure, and the runtime proactively selects one activation of grain C to join to the transaction before restarting grain A. This prevents the same inconsistency by ensuring that grains B and D will choose the same activation.

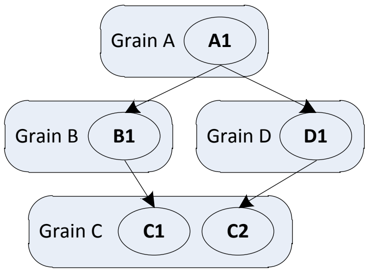

Figure 1: Consistency failure if the transaction sees two different activations (C1 and C2) of a grain

Between transactions, the Orleans consistency model ensures that the sequence of grain states visible to a requestor (whether an outside client process or another grain) always reflects its previous operations, so that a single requestor always observes its own writes. This guarantee must be maintained in the presence of quiescence, branching, and merging of activation states, in either the requestor or the target. Orleans does not provide any consistency guarantees for uncommitted changes between independent requestors. Independent requestors may ensure visibility of changes by waiting for their transactions to be committed.

3.8.3 Atomicity

To preserve atomicity, Orleans must ensure that a transaction's updates, from its set of grain activations, become visible to other transactions as a complete set or not at all. To ensure this, the runtime keeps the transaction/activation correspondence until transactions are committed (Section 3.8.4). Before joining an activation to a transaction, it verifies that this action preserves atomicity. If the active transaction has invoked the same grains as a prior, uncommitted transaction, it must use the same activations.

For example, in Figure 2 a completed transaction TX has modified activations A1, B1, and C1, and a completed transaction TY has modified D1, C2, and E1. Active transaction TZ has modified activations F1 and B1 and sends a request to grain E. If this message arrives at activation E1, 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 C. It might choose to redirect the message to another activation of grain E. Or, if none is available and it is too expensive to create a new one, it may go ahead and join activation E1 to TZ. So far, atomicity is preserved. However, if TZ does later send a message to grain C, the runtime cannot choose either activation C1 or C2 without violating atomicity (of TY or TX, respectively). The runtime will detect this before the message to grain C 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 B1 and E1 will be lost when TZ aborts.

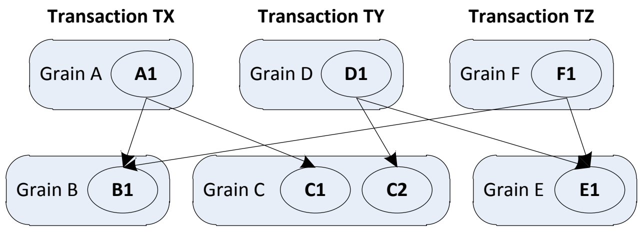

Figure 2: Potential atomicity violation; Transaction TZ cannot use either grain C1 or C2 without violating the atomicity of TY or TX, respectively

3.8.4 Durability

Orleans also ensures that committed transactions are written atomically to persistent storage. The transaction persistence mechanism also follows an optimistic approach, 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:

- The store collects serialized representations of the persistent state of each activation in the transaction.
- If more than one version of a grain is to be committed – either because a more recent version already committed to the store, or because of multiple, concurrent transactions – their state must be reconciled to produce a single merged version before writing it to store (Section 3.8.5).
- The grain states are written to persistent storage using a two-phase commit to ensure that all updates become visible simultaneously.

This process runs without blocking executing transactions, and so can fall back to an earlier phase as additional update notifications arrive for a participating grain that has already been partially processed. Our current implementation of the system store is a single point of failure and a scalability bottleneck. We are in the process of developing a distributed persistence management mechanism that will remove these limitations.

3.8.5 Reconciliation

Reconciliation occurs as application state is written to persistent storage. The reconciliation model uses a branch-and-merge model, tracking and reconciling changes to multiple independent revisions of the grain state [4]. The existing persistent state is considered the master revision. When created, each grain activation branches a new revision from the store, as illustrated in Figure 3.

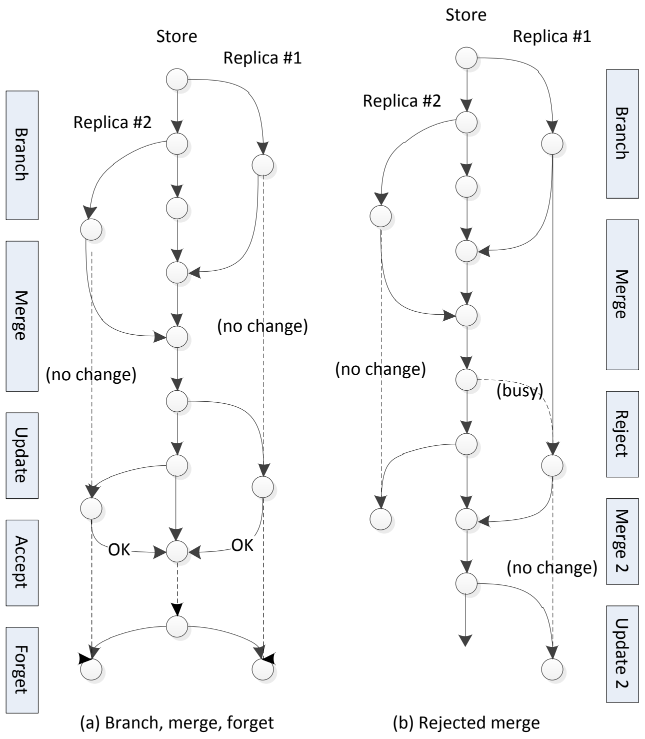

Figure* 3*: Reconciliation

When committing multiple transactions that utilized different activations of the same grain, Orleans requests the current state of each activation. These are sent as incremental deltas from the branch point to minimize data transfer and simplify reconciliation. Orleans uses a grain type and data structure-specific policy to reconcile changes to a single merged state and sends updates to each activation to bring it forward to that state. If an activation has not modified its state since sending it to the Orleans, the activation accepts the update and proceeds. If all activations accept the updates, Orleans notifies them that they have effectively returned to an initial state as freshly branched revisions from the store.

If an activation was modified, it rejects the update, and subsequently sends a second, compound set of changes to Orleans. The process then repeats. Eventually an activation will accept its updates and merge, although in a busy system the runtime may need to prevent a grain from accepting new transactions in order to ensure it does not fall too far behind.

3.8.6 Failure recovery

If a transaction encounters an error and aborts before it completes, all of its activations are destroyed and the request is re-executed. However, if some of its activations had participated in prior transactions that have completed but not committed, then the earlier transactions also must abort and re-execute since their state was not persisted.

3.8.7 Relation to Snapshot Isolation

Orleans transactions provide a model comparable to, but slightly weaker than, snapshot isolation (SI). There are two main differences between Orleans transaction and SI. First, SI does not allow write-write conflicts, while Orleans does. Orleans allows different activations of the same grain's state to be changed concurrently by independent transactions and to be later merged and reconciled into a single, application-consistent state.

Second, under SI, a transaction sees a consistent snapshot of all committed updates as of the time when the transaction starts. Future changes that occur after the transaction has started are not visible to this transaction. In Orleans, a transaction sees atomically consistent subsets of completed transactions. Those subsets become visible to the transaction at different points in time of its execution, not necessarily at its start. Every subset of completed transactions includes all grain activations changed during these transactions and all the transactions they transitively depended upon [7]. The changes made by a completed transaction become visible to future transactions atomically as a set of activations that form a consistent view. When an activation joins a running transaction, its consistent view of activations is checked for compatibility and merged with the running transaction. Unlike SI, the transaction expands its consistent view lazily as it executes, and there may be different activations of the same grain belonging to distinct consistent views in different transactions. This design, although weaker than serializability or SI, maximizes responsiveness and system throughput and does not require global coordination.

4. APPLICATIONS

We describe three applications built on Orleans to illustrate the flexibility of its architecture and programming model. The applications differ in the way they use the system. The first application is a Twitter[1]-like messaging application, which is communication-intensive with little data or computation. The second application is a linear algebra library for large sparse matrix computations, which is both computation-, communication-, and IO- intensive. The third is a distributed engine for querying and processing large graphs, which is data-intensive. The applications differ significantly in the size and number of grains and types of interactions between the grains.

4.1 Chirper

Chirper is a large-scale Twitter-like publish-subscribe system for distributing small text message updates within a large network of consumers / subscribers. It allows a user to create an account, follow other users, and receive messages posted by them on their account. We built Chirper in 200 lines of Orleans C# code. It includes only the core functionality of subscribing to a publisher and publishing and receiving messages. It does not include authentication and security, archival message storage, or message searching and filtering. Replication, persistence, and fault tolerance, however, are managed automatically by Orleans.

4.1.1 Implementing Chirper on Orleans

A Chirper account is naturally modeled as a grain. Each user has an account grain and accounts for different users act independently and interact via well-defined interfaces. An account grain has 3 facades: publisher, subscriber and account management, for different types of interactions. The account grain declares some of its properties as persistent – user id, name, list of published and received messages, a list of contacts that follow this user (subscribers), and a list of contacts this user follows (publishers). All of the persistent state is managed automatically

[1] Twitter is a trademark of Twitter, Inc. Orleans has no relationship with Twitter, Inc.

by the Orleans runtime. Some properties are declared as `InitOnly`, which indicates that they are immutable after initialization. Some properties allow querying. Other properties specify a reconciliation strategy; for example, the list of published messages is a special `SyncList` data type, which accumulates the changes in different activations and merges the additions and deletions.

The account grain exposes three public interfaces: `IChirperAccount`, `IChirperSubscriber` and `IChirperPublisher`. `IChirperAccount` represents a single user account and allows a user to start and stop following another user, retrieve the list of their followers, retrieve the list of users they follow, and retrieve the list of received messages. The user can also publish a new message via `IChirperAccount`. `IChirperSubscriber` and `IChirperPublisher` represent the view of one user on another for subscription and notification activities. When user A subscribes to user B, A invokes the `AddFollower` method of B's `IChirperPublisher` interface, passing his or her own `IChirperSubscriber` interface. When B's account has a new message, it notifies A's `IChirperSubscriber` interface. Figure 4 contains partial interfaces for Chirper.

By default, Orleans creates one activation per account grain. However, hot accounts will automatically be replicated by the runtime. This helps achieve scalability and eliminate bottlenecks. The following types of application behavior will result in creating multiple activations:

- An account that receives a large number of messages will be replicated to spread the subscription load. Each activation will receive a subset of the messages, to be merged as described in section 3.6. This allows accounts to scale with increasing received message load.
- An account that publishes a large number of messages (a chatty chirper) will be replicated to partition the publishing load. Each message will be published to all subscribers by a single activation. This will allow publishers to scale with increasing number of published messages.

There is another case in which multiple activations do not help solve the problem: an account that has an extremely large number of subscribers (a popular chirper), so that the list requires a significant portion of memory on a single server and simply iterating through the subscribers takes an appreciable amount of time. In this case the state of the account (list of subscribers) needs to be partitioned by the application. One possible solution is a hierarchical system, where the list of subscribers is partitioned into a hierarchy of small helper grains with a distribution tree from the main account grain to the helper grains which notify the subscribers. Such a pattern is easy to implement on Orleans, and would allow publishers to scale with an increasing number of subscribers.

Chirper leverages Orleans transactions to ensure that publishers and subscribers are always paired. Establishing a "following" relationship is done as follows: when user A wants to follow another user B, A sends a message to B with a reference to his subscriber grain interface and B stores the grain reference to A in its Followers list. A also stores B's identity it its Subscriptions list, so that it will be able to know later on whom A follows. The update to both grains thus needs to be done atomically, so in case of intermediate failures the transaction is re-executed, so the system is kept in a synchronized and consistent state.

```
public interface IChirperSubscriber : IGrain
{
  AsyncCompletion NewChirp(ChirperMessage chirp);
}

public interface IChirperPublisher : IGrain
{
  [Queryable(IsUnique=true)] [InitOnly]
  AsyncValue <long> UserId { get; }
  [Queryable(IsUnique=true)]
  AsyncValue <string> UserName { get; }
  [ReadOnly]
  AsyncValue<List<ChirperMessage>>
    GetPublishedMessages(int n, int start);
  AsyncCompletion AddFollower(string u,
    IChirperSubscriber s);
  AsyncCompletion RemoveFollower(string u,
    IChirperSubscriber s);
}

public interface IChirperAccount : IGrain,
  IChirperPublisher, IChirperSubscriber
{
  AsyncCompletion PublishMessage(string chirpText);
  [ReadOnly]
  AsyncValue <List<ChirperMessage>>
    GetReceivedMessages(int n, int start);
  AsyncCompletion FollowUser(string user);
  AsyncCompletion UnfollowUser(string user);
  [ReadOnly]
  AsyncValue <List<string>> GetFollowingList();
  [ReadOnly]
  AsyncValue <List<string>> GetFollowersList();
}
```

Figure 4: Chirper grain interfaces

4.2 Linear Algebra Library

Linear algebra is a broad area that comprises general-purpose computations on scalars, vectors, and matrices (including higher dimensions as tensors). The core of a linear algebra library is the vector-matrix multiplication operation. This operation is the basis for many algorithms, including PageRank, singular value decomposition, clustering, feature extraction, and social group discovery (partitioning). Conceptually, vector-matrix multiply is quite simple, and an implementation can be written very efficiently if the data set can be held in memory on one machine. As the data size grows, distributing the computation and maintaining efficiency becomes difficult due to the complexity and limitations of data placement, disk access, network bandwidth and topology, and memory limitations. A web graph, for example, may contain greater than 10^{11} pages with more than 10^{12} links; this translates to a sparse 10^{11} by 10^{11} matrix, with 10^{12} cells (out of a total of 10^{22}) having non-zero values.

Our coworker has implemented a linear algebra library on top of Orleans. The computations are broken into worker grains that own pieces of the data set. Special coordinator grains manage the computation by dynamically assigning work to worker grains. The coordinator grains are organized into a two-level hierarchy, with each second-tier grain responsible for a set of worker grains. The data can flow directly from disk to worker grains and between the worker grains, while the coordinator grains participate only in the control flow of the operations. Currently, we take advantage of explicit runtime APIs that Orleans provides to control the server placement of the grain activations; worker grains can be co-resident on the same machine (typically one per hardware thread) or distributed across many machines (co-located with the secondary storage that holds the data). In the future, we plan that many of these explicit decisions will be replaced by automated Orleans resource management (Section 7).

4.3 Large Graph Engine

Graphs are central to web search, 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 [8]. Our graph engine provides support for partitioning and distributing graph data (nodes, edges, and metadata) across many machines and for querying graphs. In contrast to the linear algebra library, where data is represented by a numerical matrix, the graph engine supports rich node and edge data types with user-defined properties and metadata, similar to database rows.

Orleans offers two options for implementing graphs: encapsulate each node in a separate grain or represent a partition of the nodes by a grain. We selected the latter approach because it allows for a significant reduction in overhead and messages because steps between grains during a graph traversal may be batched together based on the source and destination partitions. Every server hosts a small number of partition grains, and every partition grain contains a moderate number of graph data nodes (10^4 – 10^6). A graph algorithm running in a partition directly accesses nodes in its partition. Accesses across partitions involve sending messages between partition grains. The graph algorithms are aware of this distinction and batch messages between partitions to reduce communication overhead.

The graph engine demonstrates 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 engine is built upon an abstract graph execution framework similar to Pregel [8].

5. PERFORMANCE MEASUREMENTS

We measured the performance of the current Orleans implementation with a set of benchmarks. The measurements were performed on a cluster of up to 50 servers, each with two AMD Quad-Core Opteron processors running at 2.10GHz for a total of 8 cores per server, 32GB of RAM, all running 64 bit Windows Server 2008 R2.

5.1 Micro Benchmarks

Figure 5 depicts the round-trip latency of a grain method invocation, for grains located on the same and different servers. The method invocation had one parameter, a byte buffer of varying size. The latency for the remote case is approximate 1.2 millisecond, and 0.5 millisecond for the local case. In the local case, time is spend primarily in making a deep copy of the message body and in thread synchronization. The remote case adds time to serialize the message and response and their headers, socket read/write and the actual network latency. For large messages, latency increases proportionaly to the message size, due to the cost of copy and serialization.

We also measured the overhead of promises. The time to create a promise and trigger its result delegate is 50–100 microseconds, which is mainly due to memory allocation and thread synchronization. This is small compared to the message latency.

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

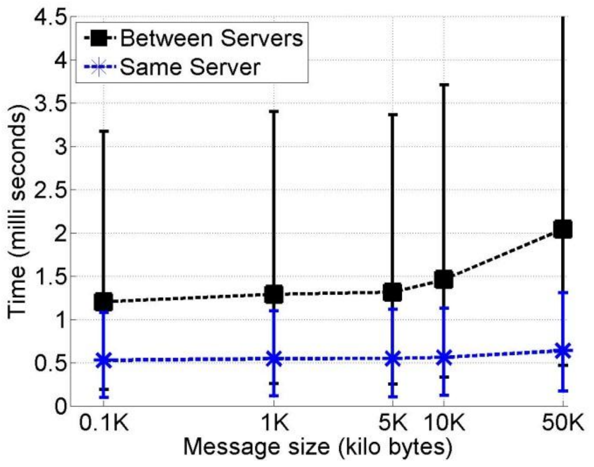

Figure 5: Local and remote invocation latency, average and 95 percentile.

5.2 Chirper

5.2.1 System throughput

In this section, we measure the performance and scalability of the Chirper application (Section 4.1). We created a synthetic network of 1,000 user accounts (we tested the application with millions of user accounts, but because the messaging throughput is insensitive to the total number of users, we measured with 1,000 to reduce the initial load time), each user following 27 random users – this corresponds to the average number of followers in the actual Twitter service. We ran load generator processes on multiple machines, with each generator posting messages of varying size, averaging 250 characters, to a random user account. Each message was then sent to the user's followers. We ran each load generator so that it generates the maximum number of messages per second that it could handle before saturation.

In this scenario, the Orleans servers running Chirper run at 94–96% CPU utilization, receiving and de-serializing messages, processing them, and serializing and resending them to follower grains. The majority of server's work is spent serializing and de-serializing messages.

Figure 6 shows that the throughput (number of chirps per second) scales linearly with the number of servers. The system can deliver approximately 7,000 chirps per second with 35 servers. By comparison, the actual Tweeter service delivers about 1600 tweets per second on average; the highest recorded single-second load was 6,939 tweets per second [9].

5.2.2 Multiple activations

In this section we show that the Orleans runtime can automatically maintain the performance of an application by creating multiple activations. We put a heavy load on a single grain by simulating a subscriber who follows many users. We also added a short processing time of 10 milliseconds for every message. Thus, this user's grain becomes a bottleneck, since it can process only a limited number of messages per second.

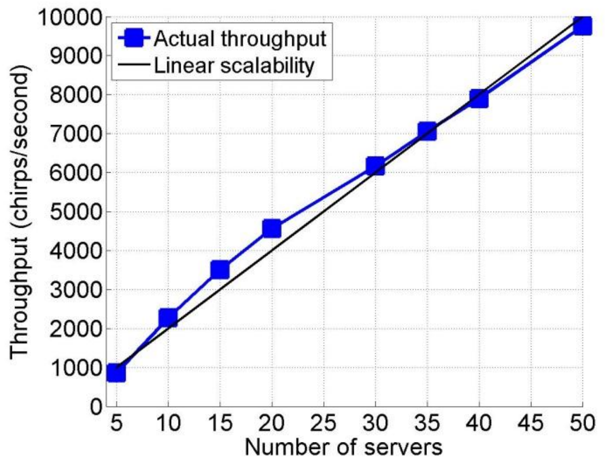

Figure 6: Chirper system-wide throughput – scalability with increased capacity

Figure 7 shows that the system throughput still scales almost linearly because Orleans runtime creates the multiple activations of the grain (on the same machine). This optimization occurs automatically. Publishers continue to send messages to the single logical grain for this user, and the runtime distributes the messages across the grain's activations, evenly spreading the load and processing time among activations running on different cores.

We also measured the overhead cost of the Orleans automated mechanisms by configured the system to not create multiple activations of a grain and disabling the transaction system. With the 10ms processing time per request, measured throughput is 13% higher with the transaction system disabled, and with 100ms of processing time, throughput is 2% better, in both cases compared to the system with transactions enabled but limited to single activations. We could not measure the mechanisms independently because the system cannot create multiple activations without the transaction system, which is necessary for consistency and reconciliation.

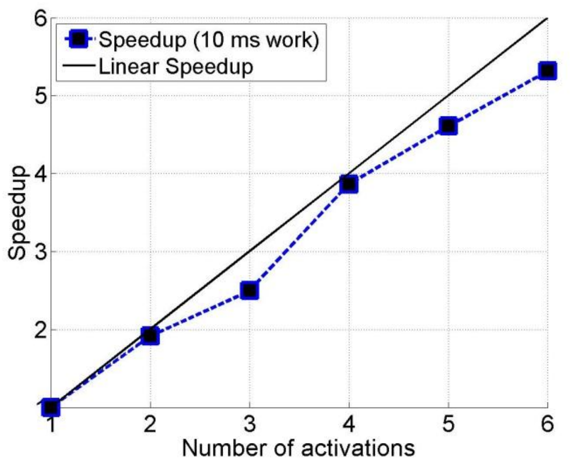

Figure 7: Maintaining Chirper throughput by automatically creating multiple activations on a single server

5.3 Linear Algebra Library

We implemented PageRank in our Linear Algebra library. The PageRank computation is essentially an iterated matrix-vector multiplication. We used a partial web graph of 134 million pages and 1.4 billon hyperlinks and ran 10 iterations of PageRank computation, until it converges. The web graph is stored in SQL databases, one per server, outside of Orleans.

Figure 8 shows the speedup on a single machine (time to compute the complete PageRank on one core vs. multiple cores). The system fails to scale well. The computation is I/O bound and thus having more cores on a single machine does not speed the computation as the machine's I/O bandwidth is totally saturated.

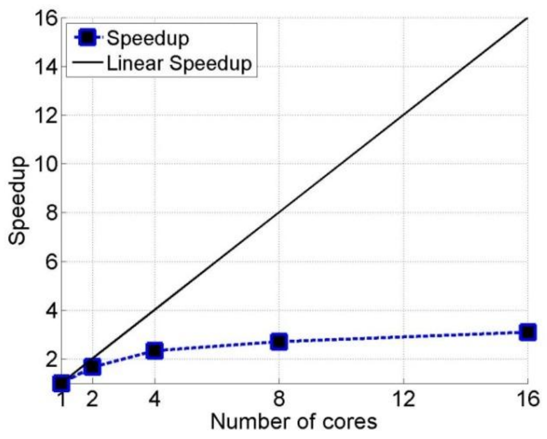

Figure 8: PageRank computation – single server, varying number of cores.

Figure 9 demonstrates the speedup on multiple machines. The speedup is much higher since the matrix is read from multiple disks, so the I/O runs concurrently with the computation. For up to 4 servers, we see near-linear speedup and for 32 machines we still get a speedup of 13. The sublinear performance is due to the increased communications overhead of exchanging data among grains (results from one iteration for the next iteration) as well as some increased coordination overhead.

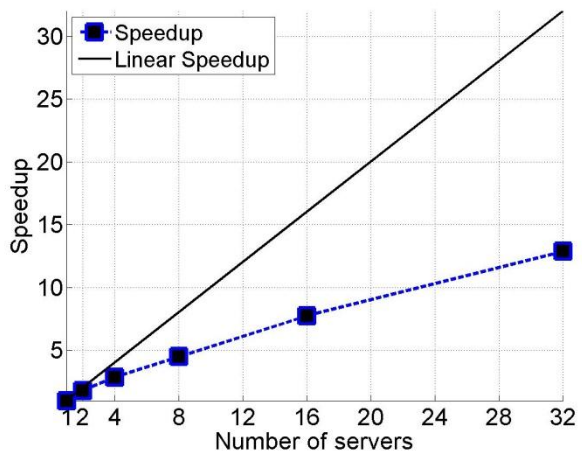

Figure 9: PageRank computation – varying number of servers, 16 cores each.

We want to stress that exactly the same program was used both in the single server and the distributed case. This demonstrates the power of Orleans to abstract the computation from its execution environment – no changes to the application were necessary to scale from 16 cores to 512 cores.

5.3.1 Other Runtimes

We also compared the Orleans implementation of PageRank against one running on Dryad [10], which provides a data-flow engine similar to map/reduce. The total computation time on Orleans was roughly two orders of magnitude faster than the Dryad implementation on the same cluster. The primary reason for this disparity is that the Orleans implementation did not write data to disk after each iteration; instead it kept data in memory inside grains and passed it other grains by direct messaging. In addition, data partitioning in Dryad is static, while Orleans permitted dynamic load balancing that accommodated the varying amount of work for different blocks in a sparse matrix. Of course, this performance gain came at the cost of increased code complexity, as the Dryad implementation is simpler and more compact.

6. RELATED WORK

Orleans is built from a combination of techniques, borrowing many concepts from previous systems. However, we believe that the combination of design choices is unique and well suited as a comprehensive solution to building scalable cloud applications.

6.1 Actors

Actors are a well-known model for concurrent programming that form the basis for many programming languages [11], including Erlang [12], E [13], Thorn[14], and many others.

Orleans extends the basic actor model with support for replication, transactions, and consistency. Replication in particular is a significant extension to the classical model, allowing Orleans systems to scale automatically when a single actor is heavily loaded. No other actor language intrinsically supports replication.

Orleans, unlike Erlang, is based on an imperative language. Moreover, Orleans communication differs from Erlang, as it is based on a single mechanism that provides expressiveness and flexibility. Promises are inherently asynchronous, like Erlang messages, but are higher-level abstractions, comparable to Erlang's synchronous RPC.

Erlang libraries support transactions and failure replication; although the strong consistency semantics is built on mechanisms less scalable than Orleans. Erlang also differs in its distributed error handling mechanism, which requires a programmer to implement guard processes to handle system failures; an approach also feasible with Orleans promises. In addition, Orleans transactions provide an automatic mechanism for recovering from system failures, going beyond Erlang's failure signaling capability.

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.

Thorn is an object-oriented, dynamic language intended to bridge the gap between exploratory scripting and production development. It provides two different communications abstractions: synchronous RPCs and explicit Erlang-style send and receive. Thorn does not provide a promise-like mechanism to unify these two abstractions, and it lacks the distributed mechanisms provided by Orleans such as replication, migration, persistence, and transactions.

6.2 Transactions

Our distributed runtime employs well-known techniques to provide service availability (replication) and data reliability (persistence). However we use a novel set of techniques for state synchronization and distributed coordination. We combine the branch-and-merge update data model [3], with application-defined reconciliation strategies, for conflict resolution with lightweight transactions for isolating computation and providing a consistent view of a distributed state.

Our distributed techniques strike a middle ground between the strong, full consistency and the weak, eventual consistency models. They provide sufficient guarantees for most applications while enabling high performance and a high degree of scalability. The traditional strong consistency model (linearizability for shared memory or serializability for transactions) provides strong guarantees for a developer, which facilitate programing and reasoning about the state of a distributed computation. However, this comes at a significant cost in performance, availability, and scalability. The weak eventual consistency model promises efficiency, availability, and scalability, but it is a complex programming model that requires a developer to explicitly reason about inconsistencies and handle with them in application code. Orleans provides a middle ground in this spectrum: each transaction sees a state that is strongly consistent with the transaction's history. Independent transactions are isolated and merge their updates via well-defined application strategies. This allows for an efficient implementation that avoids global coordination or locking while providing a much simpler programming model than eventual consistency.

6.3 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 web 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.

At a low level, Orleans differs from these in its embrace of asynchronous APIs as the programming model for all application component access. At another level, Orleans's usage of multiple activations for scalability and failure tolerance is a significant capability difference. Orleans 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.

6.4 Other

Map/reduce [6] and dataflow (Dryad) [10] frameworks, are popular for large-scale distributed computations. The map/reduce model is well-suited to off-line processing of very large data sets, but does not support interactive requests that touch a small set of related data items. We intend to incorporate standard map/reduce features such as data/processing locality into Orleans, and are investigating the possibility of implementing a map/reduce framework and programming model similar to DryadLINQ [15] on top of Orleans.

The linear algebra library demonstrates that Orleans can substitute for MPI [16]. MPI allows any general computation flow to be expressed, without the restrictions of map/reduce frameworks. Its main difference from Orleans is code complexity. MPI offers much lower level abstractions than Orleans: raw messaging, manual synchronization, no transactions and no asynchronous programming model.

7. FUTURE WORK

An on-going area of research is resource management. In Orleans, most resource management decisions revolve around activations: when and where should a new activation be created, rather than reusing an existing one? When should an existing activation be deactivated? At the next level of resources, Orleans also needs to grow and shrink the pool of server instances.

Another important area for future work is in extending Orleans to run on devices such as PCs and smartphones in order to provide a seamless programming model for cloud software across both the client and servers. Client applications raise new issues such as disconnected and intermittently connected operation, untrusted or partially trusted systems, migration between client and server, and resource management across highly heterogeneous systems.

A different, but equally important, direction is enhancing support for development and maintenance of large, long-live systems: software and API versioning, geo-distribution across data centers, and support for large data objects stored outside of Orleans.

Finally, we intend to extend the graph library (Section 4.3) and to develop additional high-level libraries for common functionality, such as pub/sub messaging and map/reduce.

8. CONCLUSIONS

This paper described the design and implementation of Orleans, a programming model and software framework for cloud computing. Orleans defines an actor-like model of isolated, replicated grains that communicate through asynchronous messages and manage asynchronous computations with promises. Isolated state and the grains' constrained execution model allow 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 the Orleans framework can significantly simplify the development of cloud applications by encouraging the use of software architectures that produce predictable, scalable, and reliable outcomes. This is not a strong guarantee, as it is possible to write a bad program in any language. Nevertheless, Orleans consciously encourages successful cloud design patterns:

- Cloud services achieve high throughput by processing multiple, independent requests concurrently. Orleans supports this style of computation by providing a programming model with strong isolation between grains, to prevent interference, and transactions spanning the grains processing a request, to ensure consistency among pieces of a single computation.
- Shared memory parallelism can reduce the latency of processing a request, but threads, locks, and concurrency are fertile sources of errors. Orleans supports a simple, single-threaded model within a grain, but permits parallelism between grains, albeit limited to message passing. In practice, nothing in the design or implementation of Orleans precludes internal parallelism, but it has not proved necessary yet.
- Computers and networks fail in distributed systems, so error-handling and recovery code is fundamental. Promises, by propagating errors equivalently with values, permit error-handling code to be concentrated in one place, much like an exception handler, rather than spread across all of the delegates invoked by promises. In addition, the transactions that wrap external requests provide the capability to roll-back and re-execute a failed computation with no explicit application code.
- Cloud applications must respond to varying and unpredictable workloads. Grain replication offers a simple, mostly transparent mechanism that permits Orleans to allocate more computing resources at bottlenecks in an application, without explicit decisions or actions by the application. Replicating a computation replicates its state and so introduces consistency problems, which Orleans handles with transactions and a multi-master, branch-and-merge update data model. Grain isolation also permits them to be migrated between servers, providing another mechanism for automatic load balancing.

Orleans is currently being used by several projects inside Microsoft Research. It is too early to report on our experience, except to note that Orleans' mechanisms and patterns are effective when used, and that training and education remains an important aspect of cloud software development.

9. REFERENCES

[1] Liskov, B. and Shrira, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, GA, June, 1988). ACM, 260-267. http://doi.acm.org/10.1145/53990.54016.

[2] Miller, M. S., Tribble, E. D. and Shapiro, J. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Proceedings of the International Symposium on Trustworthy Global Computing* (Edinburgh, UK, April, 2005). Springer, 195-229. http://www.springerlink.com/content/fu284833647hg054/.

[3] Burckhardt, S., Baldassin, A. and Leijen, D. Concurrent Programming with Revisions and Isolation Types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno, NV, October, 2010). ACM, 691-707. http://dx.doi.org/10.1145/1869459.1869515.

[4] Baker, H. G., Jr. and Hewitt, C. The Incremental Garbage Collection of Processes. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages* (August, 1977), 55-59. http://dx.doi.org/10.1145/800228.806932.

[5] Hunt, G., Aiken, M., Fähndrich, M., Hawblitzel, C., Hodson, O., Larus, J., Levi, S., Steensgaard, B., Tarditi, D. and Wobber, T. Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the 2nd*

ACM SIGOPS/EuroSys European Conference on Computer Systems (Lisbon, Portugal, March, 2007). ACM, 341-354. http://doi.acm.org/10.1145/1272996.1273032.

[6] Dean, J. and Ghemawat, S. MapReduce: a Flexible Data Processing Tool. *Communications of the ACM*, 53, 1 (January 2010), 72-77. http://doi.acm.org/10.1145/1629175.1629198.

[7] Atul, A., Barbara, L. and Patrick, O. N. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering* (San Diego, CA, February, 2000). IEEE, 67-67. http://doi.ieeecomputersociety.org/10.1109/ICDE.2000.839388.

[8] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the International Conference on Management of Data* (Indianapolis, IN, June, 2010). ACM, 135-146. http://doi.acm.org/10.1145/1807167.1807184.

[9] Twitter *Twitter Blog # numbers*. 2011. http://blog.twitter.com/2011/03/numbers.html.

[10] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal, April, 2007). ACM, 59-72. http://doi.acm.org/10.1145/1272996.1273005.

[11] Karmani, R. K., Shali, A. and Agha, G. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java* (Calgary, Candada, August, 2009). ACM, 11-20. http://doi.acm.org/10.1145/1596655.1596658.

[12] Armstrong, J. Erlang. *Communications of the ACM*, 53, 9 (September 2010), 68-75. http://doi.acm.org/10.1145/1810891.1810910.

[13] Eker, J., Janneck, J. W., Lee, E. A., Jie, L., Xiaojun, L., Ludvig, J., Neuendorffer, S., Sachs, S. and Yuhong, X. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91, 1 (January 2003), 127-144. http://dx.doi.org/10.1109/JPROC.2002.805829.

[14] Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strnisa, R., Vitek, J. and Wrigstad, T. Thorn: Bobust, Concurrent, Extensible Scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, October, 2009). ACM, 117-136. http://dx.doi.org/10.1145/1640089.1640098.

[15] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, Ú., Gunda, P. K. and Currey, J. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation* (San Diego, CA, December, 2008). USENIX Association, 1-14. http://www.usenix.org/events/osdi08/tech/full_papers/yu_y/yu_y.pdf.

[16] Snir, M., Otto, S., Huss-Lederman, S., Walker, D. and Dongarra, J. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.