Message-Passing Concurrency for Scalable, Stateful, Reconfigurable Middleware

Cosmin Arad[1,2], Jim Dowling[1,2], and Seif Haridi[1,2]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] Swedish Institute of Computer Science, Kista, Sweden
{cosmin,jdowling,seif}@sics.se

Abstract. Message-passing concurrency (MPC) is increasingly being used to build systems software that scales well on multi-core hardware. Functional programming implementations of MPC, such as Erlang, have also leveraged their stateless nature to build middleware that is not just scalable, but also dynamically reconfigurable. However, many middleware platforms lend themselves more naturally to a stateful programming model, supporting session and application state. A limitation of existing programming models and frameworks that support dynamic reconfiguration for stateful middleware, such as component frameworks, is that they are not designed for MPC.

In this paper, we present Kompics, a component model and programming framework, that supports the construction and composition of dynamically reconfigurable middleware using stateful, concurrent, message-passing components. An added benefit of our approach is that by decoupling our component execution model, we can run the same code in both simulation and production environments. We present the architectural patterns and abstractions that Kompics facilitates and we evaluate them using a case study of a non-trivial key-value store that we built using Kompics. We show how our model enables the systematic development and testing of scalable, dynamically reconfigurable middleware.

Keywords: component model, message-passing, compositional concurrency, dynamic reconfiguration, multi-core execution, reproducible simulation, distributed systems architecture.

1 Introduction

In recent times, there has been a marked increase in the use of programming languages and frameworks that support message-passing concurrency (MPC) to build high performance servers [1, 2]. The main reasons for the renewed interest in MPC are that it scales well on multi-core hardware architectures and that it provides a simple and *compositional* concurrent programming model, free from the quirks and idiosyncrasies of locks and threads. Another reason is that high performance non-blocking sockets map easily to MPC applications. In addition to this, functional programming implementations of MPC, such as Erlang [3] and Scala actors [4], have the benefit of being suitable for building middleware that is dynamically reconfigurable. Due to its stateless nature and support for

P. Narasimhan and P. Triantafillou (Eds.): Middleware 2012, LNCS 7662, pp. 208–228, 2012.

message passing, Erlang supports the construction of software that can be safely upgraded online. Message processing can be temporarily suspended in modules marked for upgrade, and the problem of transferring state from the old module to the new module is largely avoided.

The challenge we address in this paper is how to provide support for both MPC and dynamic reconfiguration in a framework for building high-performance middleware that lends itself more naturally to a stateful programming model, supporting session and application state. Existing stateful programming models and frameworks that support dynamic reconfiguration, such as component frameworks [5], are not designed for MPC support and they do not decouple their execution model from component code. As a result, they cannot run the same code in both simulation and production environments.

In previous work on dynamically reconfigurable middleware, component models, such as OpenCom [5] and Fractal [6], developed mechanisms such as explicit dependency management, component quiescence, and reconfigurable connectors for safely adapting systems online. However, the style of component interaction, based on blocking interface invocation, precludes compositional concurrency in these models making them unsuited to present day multi-core architectures.

Our work is also relevant within the context of popular non-blocking socket frameworks that are used to build high performance event-driven server applications [7], such as Lift [1] and Twitter's Finagle [8] for Scala, and Facebook's Tornado [9] for Python. Kompics' asynchronous event programming framework allows it to seamlessly integrate different non-blocking networking frameworks (such as Netty, Apache Mina, and Grizzly)[1] as pluggable components.

Kompics is a message-passing, concurrent, and hierarchical component model with support for dynamic reconfiguration. The broad goal of Kompics is to raise the level of abstraction in programming distributed systems. We provide constructs, mechanisms, architectural patterns, as well as programming, concurrency, and execution models that enable programmers to construct and compose reusable and modular distributed abstractions. We believe this is an important contribution because it lowers the cost and accelerates the development and evaluation of more reliable distributed systems. The other main motivation for our asynchronous event programming framework is performance, particularly for high-concurrency networked applications.

Through a case-study of a scalable key-value store, we show that the performance of traditional event-driven programming does not have to come at the cost of more complex programs. Using encapsulation, components can hide event-driven control flow and support component reuse. We leverage encapsulation when testing Kompics systems, by enabling the same component code to be run in both simulation and production systems. To support the easy specification of simulation experiments, we introduce a domain-specific language that provides constructs for generating simulation experiment scenarios containing thousands of nodes.

[1] `http://www.jboss.org/netty`; `http://mina.apache.org`; `http://grizzly.java.net`

A summary of our key principles in the design of Kompics are as follows. First, we tackle the increasing complexity of modern distributed systems through hierarchical abstraction. Second, we decouple components from each other to enable dynamic system evolution and runtime dependency injection. Third, we decouple component code from its executor to enable different execution environments.

2 Component Model

Kompics is a component model targeted at building distributed systems by composing protocols programmed as event-driven components. Kompics components are reactive state machines that execute concurrently and communicate by passing data-carrying typed events, through typed bidirectional ports, connected by channels. This section introduces the conceptual entities of our component model and its programming constructs, its execution model, as well as constructs enabling dynamic reconfiguration, component life-cycle and fault management.

2.1 Concepts in Kompics

The fundamental Kompics entities are events, ports, components, event handlers, subscriptions, and channels. We introduce them here and show examples of their definitions with snippets of Java code. The Kompics component model is programming language independent, however, we use Java to illustrate a formal definition of its concepts.

Events. Events are passive and immutable *typed* objects having any number of typed attributes. The type of an attribute can be any valid type in the host programming language. New event types can be defined by sub-classing old ones.

Here are two example event type definitions in Java[2]:

```
1 class Message extends Event {
2     Address source;
3     Address destination;
4 }
```

```
1 class DataMessage extends Message {
2     Data data;
3     int sequenceNumber;
4 }
```

In our Java implementation of Kompics, all event types are descendants of a root type, Event. We write DataMessage⊆Message to denote that DataMessage is a subtype of Message. In diagrams, we represent an event using the ◆Event graphical notation, where Event is the event's type, e.g., Message.

Ports. Ports are *bidirectional* event-based component interfaces. A port is a gate through which a component communicates with other components in its environment by sending and receiving events. A port allows a specific set of event types to pass in each direction. We label the two directions of a port as *positive* (+) and *negative* (−). The *type* of a port specifies the set of event types that can traverse the port in the positive direction and the set of event types that can traverse the port in the negative direction. Concretely, a port type definition

[2] We omit the constructors, getters, setters, access modifiers, and import statements.

consists of two sets of event types: a "positive" set and a "negative" set. There is no sub-typing relationship for port types.

Here are two example port type definitions in Java[3]:

```
1 class Network extends PortType {{
2     positive(Message.class);
3     negative(Message.class);
4 }}
```

```
1 class Timer extends PortType {{
2     indication(Timeout.class);      //positive
3     request(ScheduleTimeout.class);//negative
4     request(CancelTimeout.class);   //negative
5 }}
```

In this example we define a Network port type which allows events of type Message (or a subtype thereof) to pass in both ('+' and '−') directions. The Timer port type allows ScheduleTimeout and CancelTimeout events to pass in the '−' direction and Timeout events to pass in the '+' direction.

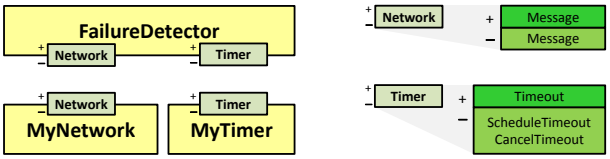

Fig. 1. The MyNetwork component has a **provided** Network port. MyTimer has a **provided** Timer port. The FailureDetector has a **required** Network port and a **required** Timer port. In diagrams, a provided port is figured on the top border, and a required port on the bottom border of a component.

Conceptually, a port type can be seen as a service or protocol abstraction with an event-based interface. It accepts *request* events and delivers *indication* or *response* events. By convention, we associate requests with the '−' direction and responses or indications with the '+' direction. In our example, a Timer abstraction accepts ScheduleTimeout requests and delivers Timeout indications. A Network abstraction accepts Message events at a sending node (*source*) and delivers Message events at a receiving node (*destination*) in a distributed system.

A component that *implements* a protocol or service will *provide* a port of the type that represents the implemented abstraction. Through this provided port, the component will receive the request events and trigger the indication events specified by the port's type. In other words, for a provided port, the '−' direction is incoming to the component and the '+' direction is outgoing from the component.

In Figure 1, the MyNetwork component provides a Network port and the MyTimer component provides a Timer port. In diagrams, we represent a port using the +/− Port graphical notation, where Port is the type of the port, e.g., Network. We represent components using the Component notation.

When a component *uses* a lower level abstraction in its implementation, it will *require* a port of the type that represents the abstraction. Through a re-

[3] The code block in the inner braces represents an "instance initializer". The *positive* and *negative* methods populate the respective sets of event types. In our implementation, a port type is a (singleton) object (for fast dynamic event filtering).

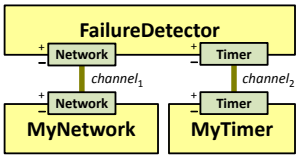

Fig. 2. $channel_1$ connects the provided Network port of MyNetwork with the required Network port of the FailureDetector. $channel_2$ connects the provided Timer port of MyTimer with the required Timer port of the FailureDetector.

quired port, a component sends out the request events and receives the indication/response events specified by the port's type, i.e., for required ports, the '−' direction is outgoing from the component and the '+' direction is incoming to the component.

Channels. Channels are first-class bindings between component ports. A channel connects two *complementary* ports of the *same* type. For example, in Figure 2, $channel_1$ connects the provided Network port of MyNetwork with the required Network port of the FailureDetector. This allows, e.g., Message events sent by the FailureDetector to be received by MyNetwork.

Channels forward events in both directions in FIFO order. In diagrams, we represent channels using the channel graphical notation. We omit the channel name when it is not relevant.

Handlers. An event handler is a first-class procedure of a component. A handler accepts events of a particular type (and subtypes thereof) and it is executed *reactively* when the component receives such events. During its execution, a handler may trigger new events and mutate the component's local state. The handlers of one component instance are *mutually exclusive*, i.e., they are executed sequentially. This alleviates the need for synchronization between different event handlers of the same component accessing the component's mutable state, which greatly simplifies their programming.

Here is an example event handler definition in Java:

```
1 Handler<Message> handleMsg = new Handler<Message>() {
2   public void handle(Message message) {
3     messages++;  // ← component-local state update
4     System.out.println("Received from " + message.source);
5 }};
```

In diagrams, we use the h⟨Event⟩ graphical notation to represent an event handler, where h is the handler's name and Event is the type of events accepted by the handler, e.g., Message.

Subscriptions. A subscription binds an event handler to one component port, enabling the handler to handle events that arrive at the component on that port. A subscription is allowed only if the handler's accepted event type, E, is allowed to pass by the port's type definition. In other words, E must be one of (or a subtype of one of) the event types allowed by the port's type definition to pass in the direction of the handler.

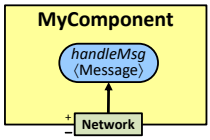

Fig. 3. The *handleMsg* event handler is **subscribed** to the required Network port of MyComponent. As a result, *handleMsg* will be executed whenever MyComponent receives a Message event on this port, taking the event as an argument.

Figure 3 illustrates the *handleMsg* handler from our previous example being subscribed to a port. In diagrams, we represent a subscription using the ⟶ graphical notation.

In this example, the subscription of *handleMsg* to the Network port is allowed because Message is in the positive set of Network; *handleMsg* will handle all events of type Message or a subtype of Message, received on this Network port.

Components. Components are event-driven state machines that execute *concurrently* and communicate *asynchronously* by message-passing. In the host programming language, components are objects consisting of any number of local state variables and event handlers. Components are modules that export and import event-based interfaces, i.e., provided and required ports. Each component is instantiated from a component definition.

Here is an example component definition in Java:

```
1 class MyComponent extends ComponentDefinition {
2   Positive<Network> network = requires(Network.class); // ← required port
3   int messages;                                        // ← local state
4   public MyComponent() {                               // ← component constructor
5     System.out.println("MyComponent created.");
6     messages = 0;
7     subscribe(handleMsg, network);
8   }
9   Handler<Message> handleMsg = new Handler<Message>() { ... };
10 }
```

In this example we see the component definition of MyComponent, illustrated in Figure 3. Line 2 specifies that the component has a required Network port. The *requires* method returns a reference to a required port, *network*, which is used in the constructor to subscribe the *handleMsg* handler to this port (line 7). The type of the required port is Positive⟨Network⟩ because, for required ports the positive direction is incoming into the component. Both a component's ports and event-handlers are first-class entities which allows for their dynamic manipulation.

Components can encapsulate subcomponents to hide details, reuse functionality, and manage system complexity. Composite components enable the control and dynamic reconfiguration of entire component ensembles as if they were single components. Composite components form a containment hierarchy rooted at a Main component (see Figure 4). Main is the first component created when the runtime system starts and it recursively creates all other sub-components. Since there exist no components outside of Main, Main has no ports.

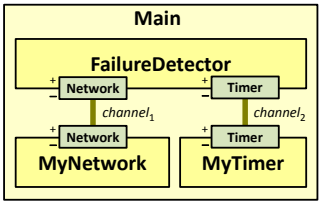

Fig. 4. The Main component encapsulates FailureDetector, MyNetwork, and MyTimer

Here is the Main component specification in Java:

```
1 class Main extends ComponentDefinition {
2   Component net, timer, fd;          // ← subcomponents
3   Channel channel1, channel2;        // ← channels
4   public Main() {                    // ↙ constructor
5     net = create(MyNetwork.class);
6     timer = create(MyTimer.class);
7     fd = create(FailureDetector.class);
8     channel1 = connect(net.provided(Network.class), fd.required(Network.class));
9     channel2 = connect(timer.provided(Timer.class), fd.required(Timer.class));
10  }
11  public static void main(String[] args) {
12    Kompics.bootstrap(Main.class);
13 }}
```

In our Java implementation, the Main component is also a Java main class (lines 11-13 show the *main* method). When executed, this will invoke the Kompics runtime system, instructing it to bootstrap, i.e., to instantiate the root component using Main as a component specification (line 12).

In lines 5-7, Main creates its subcomponents and saves references to them. In line 8, it connects MyNetwork's provided Network port to the required Network port of the FailureDetector. As a result, $channel_1$ is created and saved. Unless needed for dynamic reconfiguration (see Section 2.6), channel references need not be saved.

Components are *loosely coupled*: a component does not know the type, availability, or identity of any components with which it communicates. Instead, a component only "communicates" with its ports and it is up to the component's environment to wire up the communication.

Explicit component dependencies (required ports) enable dynamic reconfiguration of the component architecture, a fundamental feature for evolving, long-lived systems.

2.2 Kompics Operations

While presenting the Kompics concepts we have already introduced some of the basic operations on these concepts: *subscribe*, *create*, and *connect*. These have counterparts that undo their actions: *unsubscribe*, *destroy*, and *disconnect*, and these have the expected semantics. Here is the code for *destroy* and *disconnect* using our previous example:

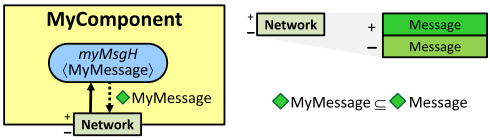

Fig. 5. MyComponent handles one MyMessage event and triggers a MyMessage reply on its required Network port

```
1 class Main extends ComponentDefinition {
2   Component net, timer, fd;          // ← subcomponents
3   Channel channel1, channel2;        // ← channels
4   public undo() {                    // ↙ some method
5     disconnect(net.provided(Network.class), fd.required(Network.class));
6     disconnect(timer.provided(Timer.class), fd.required(Timer.class));
7     destroy(net);    destroy(timer);        destroy(fd);
8 }}
```

A fundamental command in Kompics is *trigger*, which is used to (asynchronously) send an event through a port. In the next example, MyComponent handles a MyMessage event due to its subscription to its required Network port. Upon handling the first message, MyComponent triggers a MyMessage reply on its Network port and then it unsubscribes its *handleMyMsg* handler, thus handling no further messages.

Figure 5 illustrates MyComponent. In diagrams, we denote that an event handler may trigger an event on some port, using the Event graphical notation.

```
1 class MyComponent extends ComponentDefinition {
2   Positive<Network> network = requires(Network.class);
3   public MyComponent() { // ← component constructor
4     subscribe(handleMyMsg, network);
5   }
6   Handler<MyMessage> handleMyMsg = new Handler<MyMessage>(){
7     public void handle(MyMessage m) {
8       trigger(new MyMessage(m.destination, m.source), network);
9       unsubscribe(handleMyMsg, network); // ← reply only once
10 }};}
```

2.3 Publish-Subscribe Event Dissemination

Components are unaware of other components in their environment. A component can communicate, i.e., handle received events and trigger events, only through the ports visible within its scope. The ports visible in a component's scope are its own ports and the ports of its immediate sub-components. Ports and channels forward triggered events toward other connected components, as long as the types of events triggered are allowed to pass by the respective port type specifications. Hence, component interaction is dictated by the connections between components as configured by their enclosing parent component.

Component communication follows a message-passing publish-subscribe model. An event triggered (published) on a port is forwarded to other components by all channels connected to the other side of the port (Figure 6). As an optimization, our runtime system avoids forwarding events on channels that

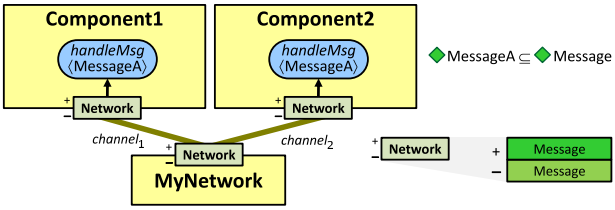

Fig. 6. When MyNetwork triggers a MessageA on its provided Network port, this event is forwarded by both $channel_1$ and $channel_2$ to the required Network ports of Component1 and Component2, respectively

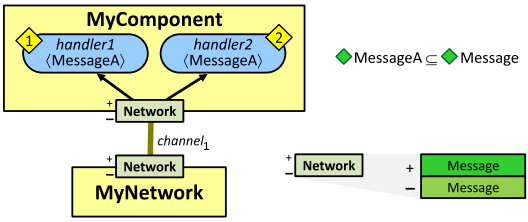

Fig. 7. When MyNetwork triggers a MessageA event on its Network port, this event is delivered to the Network port of MyComponent and handled by both *handler1* and *handler2*, sequentially (figured with yellow diamonds), in the order in which the two handlers were subscribed to the Network port

would not lead to any compatible subscribed handlers. An event received on a port is handled by all compatible handlers subscribed to that port (Figure 7).

2.4 Component Initialization and Life-Cycle

Every component provides a special Control port used for initialization, life-cyle, and fault management. Figure 8 illustrates the Control port type and a component that declares an Init, a Start, and a Stop handler. Typically, for each component definition that requires state initialization one defines a specific initialization event (subtype of Init) which contains component-specific configuration parameters.

An Init event is guaranteed to be the first event handled. When a component subscribes an Init event handler to its Control port in its constructor, the component will not handle any other event before a corresponding Init event.

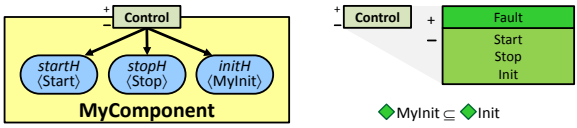

Fig. 8. Every Kompics component provides a Control port by default. To this Control port, the component can subscribe Start, Stop, and Init handlers. In general, we do not illustrate the control port in component diagrams.

```
class MyComponent extends ComponentDefinition {
  int myParameter;
  public MyComponent() { // ← component constructor
    subscribe(handleStart, control); // ← similar for Stop
    subscribe(handleInit, control);
  }
  Handler<MyInit> handleInit = new Handler<MyInit>() {
    public void handle(MyInit init) {
      myParameter = init.myParameter;
  }};
  Handler<Start> handleStart = new Handler<Start>() {
    public void handle(Start event) {
      System.out.println("started");
}};}
```

Start and Stop events allow a component (which handles them) to take some actions when the component is activated or passivated. A component is created *passive*. In the passive state, a component can receive events but it will not execute them. (Received events are stored in a port queue.) When activated, a component will enter the *active* state (executing any enqueued events). Handling life-cycle events is optional for a component.

To activate a component, a Start event is triggered on its control port, and to passivate it, a Stop event is triggered on its control port. Here is an example snippet of code possibly executed by a parent of *myComponent*:

```
trigger(new MyInit(42), myComponent.control());
trigger(new Start(), myComponent.control());
trigger(new Stop(), myComponent.control());
```

When a composite component is activated (or passivated), its subcomponents are recursively activated (or passivated). The *bootstrap* construct, introduced in the Main component example, both creates and starts the Main component.

2.5 Fault Management

Kompics enforces a fault isolation and management mechanism inspired by Erlang [3]. A software fault or exception thrown and not caught within an event handler is caught by the runtime system, wrapped into a Fault event and triggered on the Control port, as shown in Figure 9.

A composite component may subscribe a Fault handler to the control port of its subcomponents. The component can then replace the faulty subcomponent with a new instance (through dynamic reconfiguration) or take other appropriate actions. If a Fault is not handled in a parent component it is further propagated to the parent's parent and so on until it reaches the Main component. If not

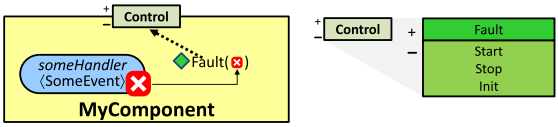

Fig. 9. Uncaught exceptions thrown in event handlers are caught by the runtime, wrapped in a Fault event and triggered on the control port

handled anywhere, ultimately, a system fault handler is executed which dumps the exception to standard error and halts the execution.

2.6 Dynamic Reconfiguration

Kompics enables the dynamic reconfiguration of the component architecture without dropping any of the triggered events. In addition to the ability to dynamically create and destroy components, connect and disconnect ports, subscribe and unsubscribe handlers, Kompics supports four channel commands to enable safe dynamic reconfiguration: *hold*, *resume*, *plug*, and *unplug*. The *hold* command puts the channel on hold. The channel stops forwarding events and starts queuing them in both directions. The *resume* command has the opposite effect, resuming the channel. When a channel resumes, it first forwards all enqueued events, in both directions, and then keeps forwarding events as ususal. The *unplug* command, unplugs one end of a channel from the port where it is connected, and the *plug* command plugs back the unconnected end to a (possibly different) port.

To replace a component *c1* with a new component *c2* (with similar ports), *c1*'s parent, *p*, puts on hold and unplugs all channels connected to *c1*'s ports; then, *p* passivates *c1*, creates *c2* and plugs the unplugged channels into the respective ports of *c2* and resumes them; *c2* is initialized with the state dumped by *c1* and activated. Finally, *p* destroys *c1*.

3 Implementation

We have implemented Kompics in Java. In this section we discuss some of the implementation details related to the runtime system, component scheduling, different modes of execution, and component dependency management. Kompics is publicly released as an open-source project. The source code for the Java implementation of the Kompics runtime, component library, and case studies presented here, are all available online at http://kompics.sics.se.

Java Runtime and Network I/O. Our Java runtime system implements the Kompics concepts and operations as well as the Kompics execution model. The Kompics runtime system supports pluggable component schedulers, decoupling component behaviour from component execution. In particular, this enables the ability to use different component schedulers to execute the same (unchanged) component-based system in different modes: parallel multi-core execution and deterministic simulation. Next subsection highlights the default scheduler.

We implemented a rich library of components and ports that provide basic distributed systems abstractions. For example, we have three different implementations for the Network abstraction using Apache MINA, Netty, and the Grizzly network library, respectively. Each of these components implements automatic connection management, message serialization, and Zlib compression. The choice of implementations is configurable - for example, CATS in section 4 uses Grizzly with Kyro for message serialization.

Multi-core Component Scheduling. The Kompics execution model admits an implementation with one lightweight thread per component. However, as Java has only heavyweight threads, we use a pool of worker threads for concurrently executing components. Every component is marked as *idle* (if it has no events awaiting execution), *ready* (if it has one or more events waiting in ports to be executed in handlers), or *busy* (if an event is currently being executed in a handler). Each worker has a dedicated queue of *ready* components. Workers process one event in one component at a time and one component cannot be processed by multiple workers at the same time. Thus, the Kompics execution model guarantees that handlers of a single component instance execute mutually exclusively.

Workers may run out of ready components to execute, in which case they engage in *work stealing* [10]. Work stealing involves a *thief*, a worker with no ready components contacting a *victim*, the worker with the highest number of ready components, and stealing a batch of half of its ready components. Stolen components are moved from the victim's work queue to the thief's work queue. From our experiments, batching shows a considerable performance improvement over stealing small numbers of ready components. To improve concurrency, the work queues are implemented as lock-free queues, meaning that the victims and thieves can concurrently consume ready components from their queues.

By designing components as reactive state machines and scheduling them using a pool of worker threads, we provide a simple programming model that leverages multi-core machines without any extra programming effort.

Deterministic Simulation Mode. We provide a special scheduler for reproducible system simulation. The system code is executed in deterministic simulation provided it does not attempt to create threads. In simulation mode, the system's bytecode (including any binary libraries) is instrumented to intercept all calls for the current time and return the simulated time. Therefore, without editing any of its source code, the system can be executed deterministically in simulated time. Library code for secure random number generators is also instrumented to use the same seed and achieve determinism. Attempts to create threads are also intercepted and the simulation halts since it would not be able to guarantee deterministic execution.

Testing and Programming in the Large. Kompics supports test-driven development through both unit-testing and integration-testing. Firstly, since components are implemented in Java classes, a component can be mocked, so that the individual handlers can be unit-tested. Secondly, integration tests (tests covering more than one component) can be implemented as Java unit tests running the tested subsystem in simulation mode, enabling systems to be built and validated using standard continuous integration platforms. To this end, we used Apache Maven to organize the structure and manage the artifacts of the Kompics component library. The complete framework counts more than 100 modules. We organize the various Kompics concepts into *abstraction* and *component* packages. An abstraction package contains a port together with the request and

indication events of that port. A component package contains the implementation of one component with some component-specific events (typically subtypes of events defined in required ports). The source code for an abstraction or component package is organized as a Maven module and the binary code is packaged into a Maven artifact, a JAR archive annotated with meta-data about the package's version, dependencies, and pointers to web repositories from where (binary) package dependencies are automatically fetched by Maven.

In general, abstraction packages have no dependencies and component packages have dependencies on abstraction packages for both the required and provided ports. This is because a component implementation will use event types defined in abstraction packages, irrespective of the fact that an abstraction is required or provided. Maven enables the reusability of protocol abstractions and component implementations. When we start a project for a new protocol implementation we just need to specify what existing abstractions our implementation depends on. They are automatically fetched and made visible in the new project. This approach also enables deploy-time composition.

4 Case Study: A Scalable, Consistent Key-Value Store

To put into perspective the Kompics concepts, patterns, and different execution modes, we present a case study of a key-value store called CATS that provides a simple API to get and put key-value pairs, while guaranteeing linearizable consistency in partially synchronous, lossy, partitionable and dynamic networks [11]. Kompics was used to develop, deploy, stress-test, and simulate CATS. This is a (non-trivial) large-scale, self-organizing distributed system with dynamic node membership. Each node in the system handles a complex mix of protocols for failure detection, topology maintenance, routing, replication, group membership, agreement, and data consistency. In the next section we highlight the component based software architecture of the system and later we show how the same system implementation designated for deployment is executed in simulation mode for debugging and testing under a wide array of concurrency and failure scenarios.

4.1 CATS Deployment Architecture

Firstly, we provide a general component framework with protocols reusable in many large-scale distributed systems. Such systems typically need a bootstrap procedure to assist newly arrived nodes in finding nodes already in the system in order to execute any join protocols. To this end, we have a BootstrapServer component which maintains a list of online nodes. Every node embeds a BootstrapClient component which provides a Bootstrap service to the node. When a node starts, it issues a BootstrapRequest to the client which retrieves from the server a list of alive nodes and delivers a BootstrapResponse to the node. The node runs a join protocol against one or more of the returned nodes and after joining, it sends a BootstrapDone event to the client, which, from now on, will send periodic keep-alives to the server letting it know this node is still alive. The BootstrapServer evicts nodes who stop sending keep-alives.

Another reusable service, is a monitoring and distributed tracing service. A client component at each node periodically inspects the status of various local components, and may also log network events for tracing. The client periodically sends reports to a monitoring server that can aggregate the status of nodes and present a global view of the system on a web page. The bootstrap and monitoring servers are illustrated in Figure 10 (left), within executable main components.

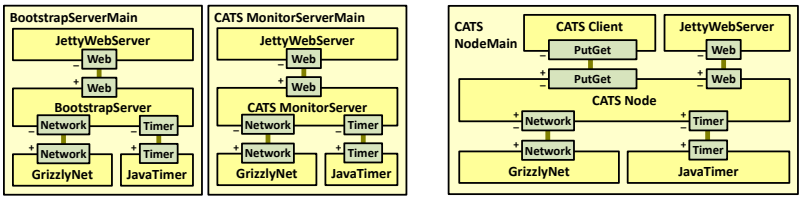

Fig. 10. Bootstrap and monitoring servers (left) exposing a user-friendly web interface for troubleshooting. Component architecture for one CATS node (right). This architecture is designated for system deployment where every CATS node executes on a different machine and communicates with other nodes by sending messages using Grizzly, a Java NIO non-blocking sockets framework.

We embed the Jetty web server library in the JettyWebServer component which wraps every HTTP request into a WebRequest event and triggers it on a required Web port. Both servers provide the Web abstraction, accepting WebRequests and delivering WebResponses containing HTML pages with the node list and global view, respectively.

In Figure 10 (right), we show the component architecture designated for system deployment. Here we have the executable CATS NodeMain component that embeds the CATS node, network, timer, web server, and client application components. The embedded CATS Node exposes its status through a Web port. The HTML page representing the node's status will typically contain hyperlinks to the neighbor nodes and to the bootstrap and monitoring server. This enables users/developers to browse the set of nodes over the web, and inspect the state of each remote node. The CATS Client component may embed a GUI or CLI user interface and issue functional requests to the CATS Node over the PutGet port.

The CATS Node is detailed in Figure 11. By encapsulating many components behind the PutGet port, clients are oblivious to the complexity and event-driven control flow internal to the component. The components used to implement CATS include a PingFailureDetector, a CATS Ring to maintain a distributed hash table, and a One-Hop Router which provides efficient message routing. The One-Hop Router, in turn, uses a service for uniform node sampling, provided by Cyclon Overlay. The Consistent ABD component provides quorum-based read and write operations, again using the One-Hop Router to find the responsible servers.

Every functional component provides a Stat port, accepting StatusRequests and delivering StatusResponses to MonitorClient and JettyWebServer. JettyWebServer enables users to monitor a node's components and issue interactive commands to the node through a web browser.

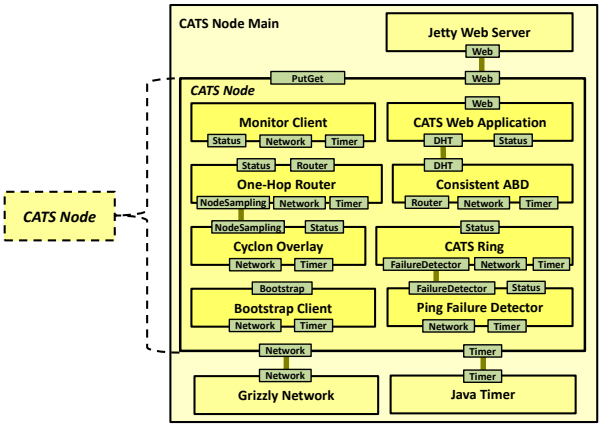

Fig. 11. The architecture of the CATS Node. We omit the channels for clarity. In this scope, all provided ports are connected to all required ports of the same type.

We have deployed and tested CATS on the PlanetLab testbed, on our local cluster, and on Rackspace. Using the web interface to interact with CATS (configured with a replication degree of 5) on the local-area network, resulted in sub-millisecond end-to-end latencies for *get* and *put* operations. This includes the LAN latency (two message round-trips, so 4 one-way latencies), message serialization (4x), encryption (4x), decryption (4x), deserialization (4x), and Kompics runtime overheads for message dispatching and execution. In terms of scalability, for read-intensive workloads, reading 1KB values, CATS scaled on Rackspace to 96 machines providing just over 100,000 reads/sec. We refer the reader to [11] for more details on CATS performance.

4.2 CATS Simulation Architecture

We now show how we can reuse the CATS' components, without modifying their code, to execute the system in simulation mode for testing, stepped debugging, or repeatable simulation studies. Figure 12 (left) shows the component architecture for simulation mode. Here, a generic NetworkEmulator/ExperimentDriver interprets an experiment scenario and issues command events to the CATS Simulator component. A command (triggered through the CATS Experiment port) may tell the CATS Simulator to create and start a new node, to stop and destroy an existing node, or to instruct an existing node to execute a system-specific operation (through its PutGet port). The ability to create and destroy node subcomponents in CATS Simulator is clearly facilitated by Kompics' support for dynamic reconfiguration and hierarchical composition. The NetworkEmulator/ExperimentDriver also provides the Network and Timer abstractions and implements a generic discrete-event simulator.

This whole architecture is executed in simulation mode, i.e., using a simulation component scheduler which executes all components that have received

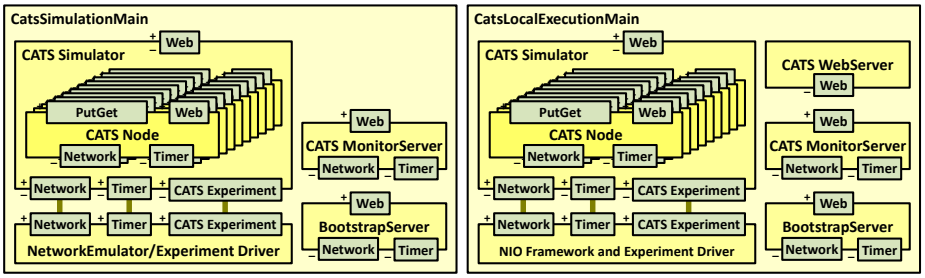

Fig. 12. Component architecture for whole-system simulation (left) / interactive stress-test execution (right). All nodes and servers execute within a single OS process in simulated time (left) / real time (right).

events and when it runs out of work it passes control to the NetworkEmulator/ExperimentDriver to advance the simulation time.

4.3 Local, Interactive, Stress-Test Execution

Using the same experiment scenario used in simulation, the same system code can be executed in an interactive stress-testing execution mode. Figure 12 (right) shows the respective component architecture. This is similar to the simulation architecture, however, our concurrent component scheduler is used and the system executes in real-time.

During development it is recommended to incrementally make small changes and quickly test their effects. The interactive execution mode helps with this routine since it enables us to quickly run a small-scale distributed system (without the need for remote deployment or launching of multiple processes) and we can interact with it using a web browser.

4.4 CATS Experimentation

We designed a Java domain-specific language (DSL) for expressing experiment scenarios for large-scale distributed systems. Such scenarios are interpreted by a NetworkEmulator/ExperimentDriver or NIO Framework and ExperimentDriver component. A scenario is a parallel and/or sequential composition of *stochastic processes*. We start each stochastic process, a finite random sequence of events, with a specified distribution of inter-arrival times.

Here is an example stochastic process:

```
1 StochasticProcess boot = new StochasticProcess() {{
2   eventInterArrivalTime(exponential(2000)); //exponentially distributed, μ = 2s
3   raise(1000, catsJoin, uniform(16)); //1000 joins with uniform IDs from 0..2^16
4 }};
```

This will generate a sequence of 1000 *catsJoin* operations, with an inter-arrival time between two consecutive operations extracted from an exponential

distribution with a mean of 2 seconds. The *catsJoin* operation is a system-specific operation with 1 parameter. In this case, the parameter is the ring identifier of the joining node, extracted from an uniform distribution of $[0..2^{16}]$. Here is how the *catsJoin* operation is defined:

```
1 Operation1<Join, BigInteger> catsJoin = new Operation1<Join, BigInteger>() {
2    public Join generate(BigInteger nodeKey){
3       return new Join(new NumericRingKey(nodeKey));
4 }};
```

It takes one BigInteger argument (extracted from a distribution) and generates a Join event (triggered by the NetworkEmulator/ExperimentDriver on PutGet port). Next, we define a *churn* process which will generate a sequence of 1000 churn events (500 joins randomly interleaved with 500 failures), with an exponential inter-arrival time with a mean of 500 milliseconds.

```
1 StochasticProcess churn = new StochasticProcess() {{
2   eventInterArrivalTime(exponential(500));//exponentially distributed, μ = 500ms
3   raise(500, catsJoin, uniform(16));       //500 joins
4   raise(500, catsFail, uniform(16));       //500 failures
5 }};
```

Next, we define a process to issues some Lookup events.

```
1 StochasticProcess lookups = new StochasticProcess() {{
2   eventInterArrivalTime(normal(50, 10));//normally distributed,μ = 50ms, σ = 10ms
3   raise(5000, catsLookup, uniform(16), uniform(14));
4 }};
```

The *catsLookup* operation takes two BigInteger parameters, extracted from a (here, uniform) distribution, and generates a Lookup event that tells CATS Simulator to issue a lookup for key *key* at the node with identifier *node*. As you can see above, a random node in $0..2^{16}$ will issue a lookup for a random key in $0..2^{14}$. 5000 lookups are issued in total, with an exponential inter-arrival time with mean 50 milliseconds.

```
1 Operation2<Lookup, BigInteger, BigInteger> catsLookup
2       = new Operation2<Lookup, BigInteger, BigInteger>() {
3    public Lookup generate(BigInteger node,BigInteger key){
4      return new Lookup(new NumericRingKey(node), new NumericRingKey(key));
5 }};
```

We defined three stochastic processes: *boot*, *churn*, and *lookups*. The next code snippet shows how we can compose them into a complete experiment scenario. The scenario starts with the *boot* process. Two seconds (simulated time) after *boot* terminates, the *churn* process starts. Three seconds after *churn* starts, the *lookups* process starts, now working in parallel with *churn*. The experiment terminates one second after all lookups are done. Putting it all together, here is how one defines and executes an experiment scenario using our Java DSL:

```
1 class CatsSimulationExperiment {
2   static Scenario scenario1 = new Scenario() {
3     StochasticProcess boot = ...  // see above
4     StochasticProcess churn = ...
5     StochasticProcess lookups = ...
6     boot.start();                                  // start
7     churn.startAfterTerminationOf(2000, boot);     // sequential composition
8     lookups.startAfterStartOf(3000, churn);        // parallel composition
9     terminateAfterTerminationOf(1000, lookups); // join synchronization
10  }
11  public static void main(String[] args) {
12    scenario1.setSeed(rngSeed);
13    scenario1.simulate(CatsSimulationMain.class); // simulation mode
14    // scenario1.execute(CatsLocalExecutionMain.class);// local, interactive
15 }}
```

Note that the above code is an executable Java main-class. It creates a *scenario1* object, sets an RNG seed, and calls the *simulate* method passing the simulation architecture of your system as an argument (line 14). The *simulate* method instruments the bytecode of the system and executes it in simulation mode, driving the simulation from the given experiment scenario. This is useful for debugging. If you want to run an interactive experiment, comment out line 14 and uncomment line 15. This will run your interactive execution architecture and drive it from the same scenario. You will be able to interact with and monitor the system over the web while the experiment is running.

Discussion and Simulation Performance. We have showed the component based software architecture of a non-trivial distributed system and how the same system implementation designated for deployment can be executed in simulation mode or interactive whole-system execution. We showed how Kompics can be used to build scalable, concurrent middleware using CATS as a case study.

We also ran simulations of CATS and we were able to simulate a system of 16384 nodes in a single 64-bit JVM with a heap size of 4GB. The ratio between the real time taken to run the simulation and the simulated time was roughly 1. For smaller system sizes we observe a much higher simulated time compression effect, as shown in Table 1.

Table 1. Time compression effects observed when simulating the system for 4275 seconds of simulated time.

Peers	Time compression
64	475x
128	237.5x
256	118.75x
512	59.38x
1024	28.31x
2048	11.74x
4096	4.96x
8192	2.01x

5 Related Work

Kompics is related to work in several areas: concurrent programming models [12, 13, 14, 15], reconfigurable component models for distributed systems [5, 6, 16], reconfigurable software architectures [17, 18, 19, 20], and event-based frameworks for distributed systems [7, 21, 22].

Kompics's message-passing concurrency model is similar to the actor model [23], of which Erlang [12], the Unix filter and pipe model, Kilim [14] and Scala [13] are, perhaps, the best known examples. Similar to the actor model, message passing in Kompics involves buffering events before they are handled in a first-in first-out (FIFO) order, thus, decoupling the thread that sends an event from the thread that handles an event. In contrast to the actor model, event buffers are associated with component ports, so each component can have more than one event queue, and ports are connected using typed channels. Channels that carry typed messages between processes are also found in other message-passing systems, such as Singularity [24]. Connections between processes in the actor models are unidirectional and based on process-ids, while channels between ports in Kompics are bi-directional and components are oblivious to the destination of their events.

The main features of the Kompics component model, such as the ability to compose components, support for strongly-typed interfaces, and explicit dependency management using ports, are found in many existing component models, such as ArchJava [20], OpenCOM [5], Fractal [6], LiveObjects [16], and OMNnet++[25]. However, with the exception of LiveObjects, these component models are inherently client-server models, with blocking RPC interfaces.

LiveObjects has the most similar goals to Kompics of supporting encapsulation and composition of distributed protocols. Its endpoints are similar to our ports, providing bi-directional message-passing, however, endpoints in LiveObjects support only one-to-one connections. Other differences with Kompics include: the lack of a concurrency model beyond shared-state concurrency, the lack of reconfigurability, and the lack of support for hierarchical components.

Although there is support for dynamic reconfiguration in some actor-based systems, such as Erlang, Kompics's reconfiguration model is based on reconfiguring strongly typed connections between components. Component-based systems that support similar runtime reconfiguration functionality use either reflective techniques, such as OpenCOM [5], or dynamic software architecture models, such as Fractal [6], Rapide [17], and ArchStudio4/C2 [19]. Kompics's reconfiguration model is most similar to the dynamic software architecture approaches, but a major difference is that the software architecture in Kompics is not specified explicitly in an architecture definition language, rather it is implicitly constructed at runtime.

Other work related to Kompics are non-blocking socket frameworks that support asynchronous event programming, such as Tornado for Python [9] and Lift for Scala actors. Protocol composition frameworks, such as Horus [26], Appia [22] and Mace [27], are also related, but they are specifically designed for building distributed systems by layering modular protocols. Although this approach

certainly simplifies the task of programming distributed systems, these frameworks are often designed with a particular protocol domain in mind and this limits their generality. Mace, however, also supports the execution of the same code in both production and simulation. Finally, there are related tools for monitoring distributed systems, such as Dapper [28] by Google, a distributed tracing system that is built-in to a few key modules commonly linked by all applications. In contrast, in Kompics, we have a monitoring client that execute concurrently and can be easily adapted to handle events published by any component.

6 Conclusions and Future Work

In this paper we presented the Kompics component model and programming framework. We showed how complex distributed systems can be built by composing simple protocols. Protocol abstractions are programmed as event-driven, message-passing concurrent components. Kompics contributes a unique combination of features well suited for the development and testing of large-scale, long-lived distributed systems, including: hierarchical component composition, dynamic reconfiguration, message-passing concurrency, publish-subscribe non-blocking component interaction, seamless integration of NIO frameworks, and the ability to run the same code in either production mode or reproducible simulation for testing and stepped debugging. For future work, we are investigating a Kompics front-end in Scala. This would immediately leverage the existing Java components and runtime system. Also, it has the potential for more expressive code and a succint DSL for Kompics operations.

References

1. Chen-Becker, D., Weir, T., Danciu, M.: The Definitive Guide to Lift: A Scala-based Web Framework. Apress, Berkely (2009)
2. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax, 1st edn. O'Reilly Media, Inc. (2010)
3. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD Dissertation, The Royal Institute of Technology, Sweden (2003)
4. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)
5. Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., Sivaharan, T.: A generic component model for building systems software. ACM Trans. Comput. Syst. 26(1), 1–42 (2008)
6. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. Softw. Pract. Exper. 36(11-12), 1257–1284 (2006)
7. Welsh, M., Culler, D., Brewer, E.: Seda: an architecture for well-conditioned, scalable internet services. In: SOSP 2001, pp. 230–243. ACM, New York (2001)
8. Wampler, D.: Scala web frameworks: Looking beyond lift. IEEE Internet Computing 15, 87–94 (2011)
9. Dory, M., Parrish, A., Berg, B.: Introduction to Tornado. O'Reilly Media (2012)
10. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)

11. Arad, C., Shafaat, T.M., Haridi, S.: CATS: Linearizability and partition tolerance in scalable and self-organizing key-value stores. Technical Report T2012:04, Swedish Institute of Computer Science (2012)
12. Armstrong, J.: Programming Erlang. In: Pragmatic Bookshelf (July 2007)
13. Odersky, M., Zenger, M.: Scalable component abstractions. In: OOPSLA 2005, pp. 41–57. ACM, New York (2005)
14. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Dell'Acqua, P. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
15. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-Safe Eventful Sessions in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 329–353. Springer, Heidelberg (2010)
16. Ostrowski, K., Birman, K., Dolev, D., Ahnn, J.H.: Programming with Live Distributed Objects. In: Dell'Acqua, P. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 463–489. Springer, Heidelberg (2008)
17. Luckham, D.C., Vera, J.: An event-based architecture definition language. IEEE Trans. Softw. Eng. 21(9), 717–734 (1995)
18. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)
19. Dashofy, E.M., Asuncion, H.U., Hendrickson, S.A., Suryanarayana, G., Georgas, J.C., Taylor, R.N.: Archstudio 4: An architecture-based meta-modeling environment. In: ICSE Companion, pp. 67–68 (2007)
20. Aldrich, J., Notkin, D.: Architectural Reasoning in ArchJava. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 334–367. Springer, Heidelberg (2002)
21. Krohn, M., Kohler, E., Kaashoek, M.F.: Events can make sense. In: USENIX ATC 2007, pp. 7:1–7:14. USENIX Association, Berkeley (2007)
22. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems, Phoenix, Arizona, pp. 707–710. IEEE (2001)
23. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
24. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in Singularity OS. SIGOPS Oper. Syst. Rev. 40(4), 177–190 (2006)
25. Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: Simutools 2008 (2008)
26. van Renesse, R., Birman, K.P., Maffeis, S.: Horus: a flexible group communication system. Commun. ACM 39(4), 76–83 (1996)
27. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. SIGPLAN Not. 42(6), 179–188 (2007)
28. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc. (2010)