



Kompics: A Message-Passing Component Model for Building Distributed Systems¹

SICS Technical Report T2010:04
ISSN 1100-3154

Cosmin Arad
icarad@kth.se

Software and Computer Systems, ICT
Royal Institute of Technology (KTH)

Seif Haridi
seif@sics.se

Computer Systems Laboratory
Swedish Institute of Computer Science

June 1, 2010

Abstract

The Kompics component model and programming framework was designed to simplify the development of increasingly complex distributed systems. Systems built with Kompics leverage multi-core machines out of the box and they can be dynamically reconfigured to support hot software upgrades. A simulation framework enables deterministic debugging and reproducible performance evaluation of unmodified Kompics distributed systems.

We describe the component model and show how to program and compose event-based distributed systems. We present the architectural patterns and abstractions that Kompics facilitates and we highlight a case study of a complex distributed middleware that we have built with Kompics. We show how our approach enables systematic development and evaluation of large-scale and dynamic distributed systems.

¹This work has been partially funded by the European Commission, IST Project SELFMAN (contract 34084) and by the Swedish Research Council (contract 2009-4299).

Contents

1	Introduction	1
2	Component model	2
2.1	Concepts in Kompics	2
2.2	Programming constructs	8
2.3	Publish-subscribe event dissemination	9
2.4	Component initialization and life-cycle	10
2.5	Fault management	11
2.6	Client-server component interaction	12
2.7	Dynamic reconfiguration	13
3	Patterns and abstractions	13
3.1	Distributed message passing	13
3.2	Event interception	14
3.3	Setting and canceling timers	15
3.4	Remote service invocation	16
3.5	Higher-level abstractions	16
4	Implementation	16
4.1	Java runtime engine and framework	16
4.2	Multi-core component scheduling	17
4.3	Deterministic simulation mode	17
4.4	Programming in the large	17
5	Case study	18
5.1	Peer-to-peer systems architecture	18
5.2	Peer-to-peer system experimentation	22
6	Related work	24
7	Conclusions and future work	26

1 Introduction

A large and increasing fraction of the world’s computer systems are distributed. Distribution is employed to achieve scalability, fault-tolerance, or it is just an artifact of the geographical separation between the system participants. Distributed systems have become commonplace, operating across a wide variety of environments from large data-centers to mobile devices, and offering an ever richer combination of services and applications to more and more users.

All distributed systems share inherent challenges in their design and implementation. Often quoted challenges stem from concurrency, node dynamism, or asynchrony. We argue that today, there is an under-acknowledged challenge that restrains the development of distributed systems. The increasing complexity of the concurrent activities and reactive behaviors in a distributed system is unmanageable by today’s programming models and abstraction mechanisms.

Any first-year computer science student can quickly and correctly implement a sorting algorithm in a general purpose programming language. At the same time, the implementation of a distributed consensus algorithm can be time consuming and error prone, even for an experienced programmer who has all the required expertise. Both sorting and distributed consensus are basic building blocks for systems, so why do we witness this state of affairs? Because currently, programming distributed systems is done at a too low level of abstraction. Existing programming languages and models are well suited for programming local, sequential abstractions, like sorting. However, they are ill-equipped with mechanisms for programming high-level distributed abstractions, like consensus.

With Kompics we aim 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.

Protocol composition frameworks exist [Isis, Horus, Amoeba, Transis, Totem, Arjuna, Bast, Psync, and Appia] and they are specifically designed for building distributed systems by layering modular protocols. This approach certainly simplifies the task of programming distributed systems, however, these frameworks are often designed with a particular protocol domain in mind and this limits their generality.

More general programming abstractions, e.g. hierarchical components, are supported by modern component models like OpenCom [8] or Fractal [6], which also provide dynamic system reconfiguration, an important feature for evolving or self-adaptive systems. However, the style of component interaction, based on synchronous interface invocation, precludes compositional concurrency in these models making them unfit for present day multi-core architectures.

Message-passing concurrency has proved not only to scale well on multi-core hardware architectures but also to provide a simple and *compositional* concurrent programming model, free from the quirks and idiosyncrasies of locks and threads.

With Kompics we propose a message-passing, concurrent, and hierarchical component model with support for dynamic reconfiguration. We also propose a me-

thodology for designing, programming, composing, deploying, and evaluating distributed systems. 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.

This paper is organized as follows. Section 2 describes the Kompics component model and illustrates its programming constructs. Section 3 presents the architectural patterns and abstractions supported by Kompics. Section 4 discusses implementation details and component scheduling in different modes of execution. Section 5 presents a peer-to-peer middleware case study, its component design and component architecture for deployment and simulated and interactive experimentation. We survey related work in Section 6 and discuss future work and conclude in Section 7.

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 conceptual entities in Kompics 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.

Code 1 shows two example event type definitions in Java. (We omit the constructors, getters, setters, access modifiers, and import statements for the sake of clarity.)


In our Java implementation of Kompics, all event types are descendants of a root event type, **Event**. Here you can see the definition of a **Message** event type having a *source* and a *destination* attribute, both of type **Address**. The **DataMessage** event type is a subtype of the **Message** event type, extending it with two more attributes: *data* of some type **Data** and an integer *sequenceNumber*. We denote the fact that **DataMessage** is a subtype of **Message** by $\text{DataMessage} \subseteq \text{Message}$.

```

1 class Message extends Event {
2     Address source;
3     Address destination;
4 }
5 class DataMessage extends Message {
6     Data data;
7     int sequenceNumber;
8 }

```

Code 1: *Example event type definitions.*

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 acts as a filter for the events that pass through it in each direction. It allows a specific set of event types to pass and disallows all other event types. 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 consists of two sets of event types: a “positive” set and a “negative” set. There is no sub-typing relationship for port types.

Code 2 shows two example port type definitions in Java².

```

1 class Network extends PortType {{
2     positive(Message.class);
3     negative(Message.class);
4 }}
5 class Timer extends PortType {{
6     request(ScheduleTimeout.class); // same as negative(...);
7     request(CancelTimeout.class);   // same as negative(...);
8     indication(Timeout.class);      // same as positive(...);
9 }}

```

Code 2: *Example port type definitions.*

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.

Conceptually, you can view a port type as a service or protocol abstraction with an event-based interface. This protocol abstraction 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,

²The 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).

a **Timer** abstraction accepts **ScheduleTimeout** requests and delivers **Timeout** indications. A **Network** abstraction accepts **Message** events at a sending node (*source*) and delivers these **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 $^+ \text{Port}$ graphical notation, where **Port** is the type of the port, e.g., **Network**. We represent components using the **Component** notation.

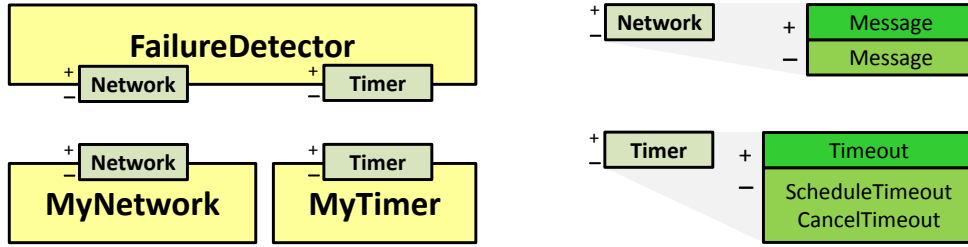


Figure 1: The **MyNetwork** component has a *provided* **Network** port. **MyTimer** has a *provided* **Timer** port. The **FailureDetector** has a *required* **Network** port and a *required* **Timer** port. Typically, a provided port is drawn on the top border, and a required port on the bottom border of a component.

When a component *uses* a lower level abstraction in its implementation, it will *require* a port of the type that represents the used abstraction. Through this required port, the component will send out the request events and receive the indication or response events specified by the port's type. In other words, for a required port, the '−' direction is outgoing from the component and the '+' direction is incoming to the component.

In Figure 1, the **FailureDetector** component requires a **Network** port and a **Timer** port. Therefore, it is possible for the **FailureDetector** to communicate with the **MyNetwork** and **MyTimer** components as the directions of their ports are compatible. For example, the **FailureDetector** could send a **ScheduleTimeout** request to the **MyTimer** and this could later send back a **Timeout** indication. To enable this communication, however, the provided and required ports of the two components would have to be connected.

Channels Channels are 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.

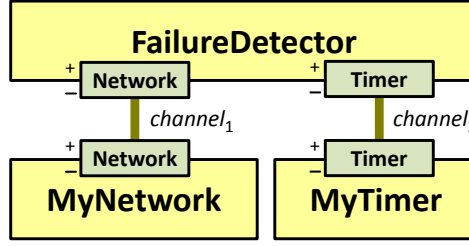


Figure 2: $channel_1$ connects the provided **Network** port of **MyNetwork** with the required **Network** port of the **FailureDetector**. $channel_2$ connects the provided **Timer** port of **MyTimer** with the required **Timer** port of the **FailureDetector**.

In diagrams, we represent a channel using the $\xrightarrow{\text{channel}}$ graphical notation, where *channel* is the name of the channel. We omit showing 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 executes *reactively* when the component receives such events. During its execution, a handler may trigger new events, mutate the component’s local state, etc. The handlers of one component are *mutually exclusive*!

Code 3 shows an example event handler definition in Java.

```

1 Handler<Message> handleMsg = new Handler<Message>() {
2   public void handle(Message msg) {
3     messages++; // ← component-local state update
4     System.out.println("Received from " + msg.source);
5   }
6 };

```

Code 3: *Example event handler definition.*

In diagrams, we use the $h(\text{Event})$ graphical notation to represent an event handler, where *h* is the handler’s name and **Event** is the handler’s type of accepted events, e.g., **Message**.

Subscriptions A subscriptions binds an event handler to one component port, allowing the handler to handle events that arrive at the component on that port. A subscription is legal if and only if the handler’s accepted event type is allowed to pass by the port’s type definition. In other words, the handler’s accepted event type 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.

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

In this example, the subscription of *handleMsg* to the **Network** port is legal, 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.

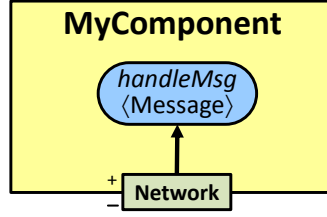


Figure 3: The `handleMsg` event handler is **subscribed** to the required `Network` port of `MyComponent`. As a result, `handleMsg` will be executed whenever `MyComponent` receives a `Message` event on this port, taking the event as an argument.

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, respectively. Each component has a constructor that is executed when the component is created. Each component is instantiated from a component definition.

Code 4 shows an example component definition in Java.

```

1 class MyComponent extends ComponentDefinition {
2   Positive<Network> network = requires(Network.class);
3   int messages; // ← local state,   ↗ required port
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    public void handle(Message msg) {
11      messages++; // ← component-local state update
12      System.out.println("Received from " + msg.source);
13    }
14  }

```

Code 4: Example component definition.

In this example we see the component definition of `MyComponent` which was 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 and manage system complexity. Composite components form component hierarchies rooted at a `Main` component. `Main` is the first component created when the runtime system starts. `Main` will create other functional sub-components. Since there exist no components outside of `Main`, it makes no sense for `Main` to have any ports.

Code 5 shows the `Main` component specification in Java.

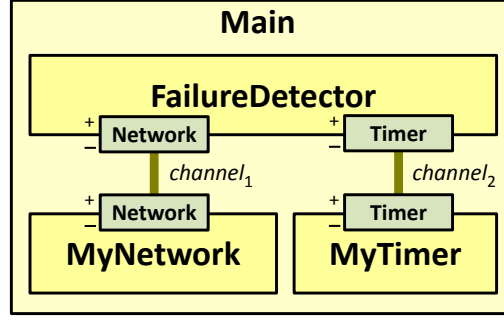


Figure 4: The *Main* component encapsulates a *FailureDetector*, a *MyNetwork* and a *MyTimer* component. *Main* is the root component therefore it can have no ports.

```

1 class Main extends ComponentDefinition {
2   Component myNet, myTimer, fd;      // ← subcomponents
3   Channel channel1, channel2;        // ← channels
4   public Main() {                    // ✓ constructor
5     myNet = create(MyNetwork.class);
6     myTimer = create(MyTimer.class);
7     fd = create(FailureDetector.class);
8     channel1 = connect(myNet.provided(Network.class),
9                       fd.required(Network.class));
10    channel2 = connect(myTimer.provided(Timer.class),
11                      fd.required(Timer.class));
12  }
13  public static void main(String[] args) {
14    Kompics.createAndStart(Main.class);
15  }
16 }

```

Code 5: Example definition of a *Main* component.

In our Java implementation, the *Main* component is also a Java main-class (lines 13-15 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 14).

In lines 5-7, *Main* creates its subcomponents and saves references to them. In lines 8-9, it connects *MyNetwork*'s provided *Network* port to the required *Network* port of the *FailureDetector*. As a result, *channel1* is created and saved. Unless needed for dynamic reconfiguration (see Section 2.7), channel references need not be saved.

Components are *loosely coupled* in the sense that 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 the dynamic reconfiguration of the component architecture, an important feature for evolving and self-adaptive systems.

2.2 Programming constructs

We have presented the fundamental Kompics concepts. Let us now turn to the programming constructs that operate on these concepts. Some of these constructs you have already met in the previous examples: *subscribe*, *create*, and *connect*. These constructs have counterparts to undo their actions: *unsubscribe*, *destroy*, and *disconnect*, and they have the expected semantics. In Code 6 you see the code for the *destroy* and *disconnect* constructs using our previous example.


```
1 class Main extends ComponentDefinition {
2   Component myNet, myTimer, fd;      // ← subcomponents
3   Channel channel1, channel2;        // ← channels
4   public undo() {                    // ✓ some method
5     disconnect(myNet.provided(Network.class),
6               fd.required(Network.class));
7     disconnect(myTimer.provided(Timer.class),
8               fd.required(Timer.class));
9     destroy(myNet);
10    destroy(myTimer);
11    destroy(fd);
12  }
13 }
```

Code 6: *Example usage of the destroy and disconnect constructs.*

A very important command in Kompics is *trigger* which allows a component to trigger an event on one of its ports. Here we have an example where `MyComponent` handles a `MyMessage` event due to its initial subscription to its required `Network` port. Upon handling the first message, `MyComponent` triggers a `MyMessage` reply (reversing *msg*'s *source* and *destination*) on its `Network` port and then it unsubscribes its *myMsgH* handler, handling no further messages. This is illustrated in Code 7.

```
1 class MyComponent extends ComponentDefinition {
2   Positive<Network> network = requires(Network.class);
3   public MyComponent() {              // ← component constructor
4     subscribe(myMsgH, network);
5   }
6   Handler<MyMessage> myMsgH = new Handler<MyMessage>() {
7     public void handle(MyMessage msg) {
8       trigger(new MyMessage(msg.destination, msg.source), network);
9       unsubscribe(myMsgH, network); // ← reply only once
10    }
11  };
```

Code 7: *Example usage of the trigger and unsubscribe constructs.*

Figure 5 illustrates `MyComponent`. In diagrams, we represent the fact that an event handler may trigger an event on some port, using the  graphical notation, where `Event` is the type of the event being triggered.

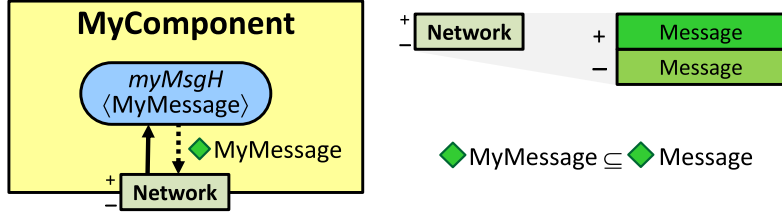


Figure 5: *MyComponent* handles one *MyMessage* event and triggers a *MyMessage* reply on its required *Network* port.

2.3 Publish-subscribe event dissemination

Components are unaware of other components in their environment. A component can only handle events that it receives on its ports and can trigger new events on its ports. The 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 specification. Hence, the component interaction is dictated by the connections between sibling components configured by their parent component.

Component interaction follows a publish-subscribe model. For example, in Figure 6, every *MessageA* event triggered by *MyNetwork* on its provided *Network* port is delivered both at *Component1* and *Component2*, by *channel₁* and *channel₂*.

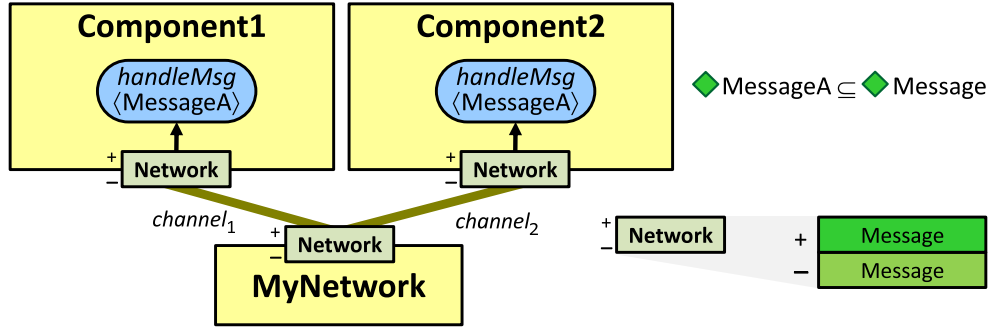


Figure 6: When *MyNetwork* triggers a *MessageA* on its provided *Network* port, this event is forwarded by both *channel₁* and *channel₂* to the required *Network* ports of *Component1* and *Component2*, respectively.

In Figure 7, however, *MessageA* events triggered by *MyComponent* are only going to be delivered at *Component1* while *MessageB* events triggered by *MyComponent* are only going to be delivered at *Component2*.

In Figure 8, whenever *MyNetwork* triggers a *MessageA* event on its *Network* port, this event is delivered to *MyComponent* where it is handled by *handler1*. Conversely, whenever *MyNetwork* triggers a *MessageB* event on its *Network* port, this event is delivered to *MyComponent* where it is handled by *handler2*.

In Figure 9, whenever *MyNetwork* triggers a *MessageA* event on its *Network* port, this event is delivered to *MyComponent* where it is handled sequentially by both *handler1* and *handler2*, in the same order in which these two handlers were subscribed to the *Network* port.

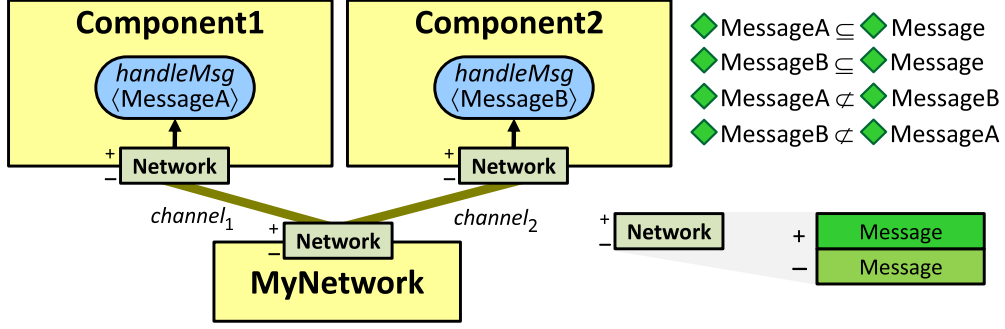


Figure 7: When MyNetwork triggers a MessageA event on its provided Network port, this event is forwarded by channel₁ to the required Network port of Component1, only. MessageB events triggered by MyNetwork on its Network port, are forwarded by channel₂ to the Network port of Component2 only.

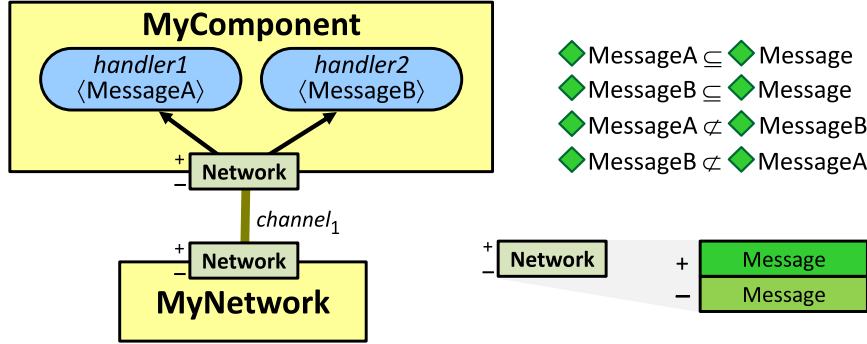


Figure 8: MessageA events triggered by MyNetwork on its Network port, are delivered to the Network port of MyComponent and handled by handler1. MessageB events triggered by MyNetwork on its Network port, are delivered to the Network port of MyComponent and handled by handler2.

2.4 Component initialization and life-cycle

Component constructors take no arguments, so to initialize a component with some configuration parameters you use a special `Init` event which you trigger on the component's `Control` port, a special port provided by every component.

Figure 10 illustrates the `Control` port type and a component that declares an `Init`, a `Start`, and a `Stop` handler. Typically, for each component you define a specific initialization event (as a subtype of `Init`) which contains component-specific configuration parameters. Code 8 shows an example `Init` event handler definition.

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.

`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

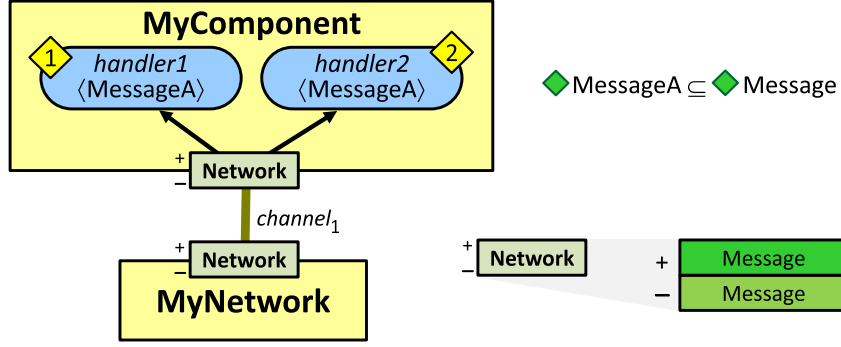


Figure 9: When **MyNetwork** triggers a **MessageA** event on its **Network** port, this event is delivered to the **Network** port of **MyComponent** and handled by both **handler1** and **handler2**, sequentially (figured with yellow diamonds), in the order in which the two handlers were subscribed to the **Network** port.

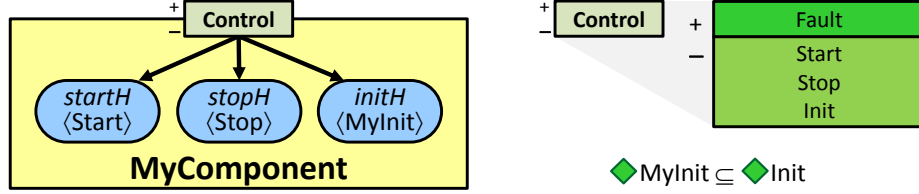


Figure 10: Every *Kompics* component provides a **Control** port by default. To this **Control** port, the component can subscribe **Start**, **Stop**, and **Init** handlers. In general, we do not illustrate the control port in component diagrams.

events is optional for a component.

You activate a component by triggering a **Start** event on its control port, and you passivate it by triggering a **Stop** event. Code 9 shows an example snippet of code possibly executed by a parent of *myComponent*.

When a composite component is activated (or passivated), its subcomponents are recursively activated (or passivated). The *createAndStart* 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]. An exception or software fault 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 11.

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 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 the **Main** component. If not handled anywhere, ultimately, a system fault handler is executed which dumps the exception to standard error and halts the execution.

```

1 class MyComponent extends ComponentDefinition {
2     int myParameter;
3     public MyComponent() {          // ← component constructor
4         subscribe(startH, control); // ← similar for Stop
5         subscribe(initH, control);
6     }
7     Handler<MyInit> initH = new Handler<MyInit>() {
8         public void handle(MyInit init) {
9             myParameter = init.myParameter;
10        };
11    Handler<Start> startH = new Handler<Start>() {
12        public void handle(Start event) {
13            System.out.println("started");
14        };
15    }

```

Code 8: *Example Init and Start handlers in a component definition.*

```

1 trigger(new Start(), myComponent.control());
2 trigger(new Stop(), myComponent.control());
3 trigger(new MyInit(42), myComponent.control());

```

Code 9: *Triggering Start, Stop, and Init events on the Control port of a subcomponent.*

2.6 Client-server component interaction

Assume you have a *server* component, i.e., a component providing a service with an interface based on requests and responses (e.g., Timer). Further assume you have multiple instances of a *client* component, using the service. Given the publish-subscribe semantics described in Section 2.3, what happens is that when one of the clients issues a request and the server handles it and issues a response, all clients receive the response. For this situation, Kompics provides two special types of events: **Request** and **Response**. These should be used in any port type definition which represents a request-response service with potentially multiple clients.

When a **Request** event is triggered by a client, as the event passes through different channels and ports in the architecture, it saves them on a stack. When the server generates a **Response** event, it borrows the **Request**'s stack. As the **Response** event passes through the architecture it pops its stack one element at a time to see where to go next. This way, only the initiator client will get the **Response** event.

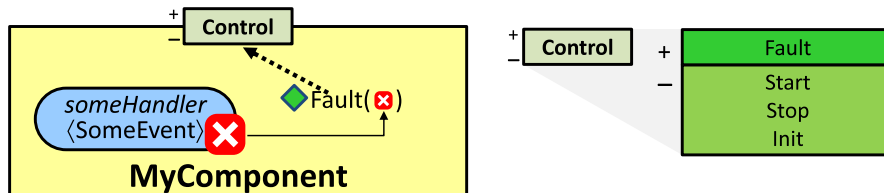


Figure 11: *Uncaught exceptions thrown in event handlers are caught by the runtime, wrapped in a **Fault** event and triggered on the control port.*

2.7 Dynamic reconfiguration

Kompics supports dynamic reconfiguration of the component architecture without loss of events. We highlight here the most common type of reconfiguration operation: swapping a component instance with a new instance.

Channels support four 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 usual. 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* (of the same type), one passivates *c1* and puts on hold all channels connected to *c1*'s ports. These channels are then unplugged from the ports of *c1* and plugged into the respective ports of *c2*. *c1* is instructed to dump its state into XML and *c2* is initialized using the dumped state of *c1*. All channels now connected to *c2* are resumed, *c2* is activated and *c1* is destroyed.

3 Patterns and abstractions

Now that you are familiar with the basics of the component model let us look at a few of the idioms, patterns, and abstractions supported in Kompics.

3.1 Distributed message passing

You send messages between remote nodes in a distributed system using the **Network** abstraction. Typically, when you implement a protocol as a component, you define component-specific protocol messages as subtypes of the **Message** event.

In Figure 12 you see two processes sending **Ping** and **Pong** messages to each other as part of a protocol implemented by **MyComponent**. When we designed **MyComponent** we knew we needed to handle **Ping** and **Pong** messages, so we defined these message types so we could subscribe *pingH* and *pongH* for them. Being subtypes of **Message**, both **Ping** and **Pong** have *source* and *destination* **Address** attributes. When **MyComponent** in **Main1** wants to send a ping to **MyComponent** in **Main2**, it creates a **Ping** message using its own address as source, and **Main2**'s address as destination, and triggers it on its required **Network** port. This **Ping** event is handled by **MyNetwork** in **Main1**, which marshals it and sends it to **MyNetwork** in **Main2**, which unmarshals it and triggers it on its **Network** port. The **Ping** event is delivered to **MyComponent** in **Main2** where it is handled by *pingH*.

The **MyTimer** component is configured in each process with a network address it should listen on for incoming connections. **MyNetwork** automatically manages network connections between processes. Each message type can optionally set a *transport* attribute which can be UDP or TCP (default). **MyNetwork** will send the message on a connection of the desired type.

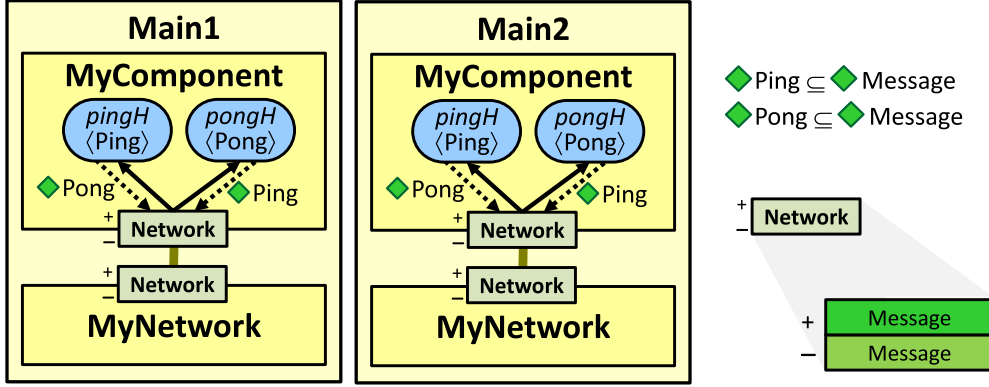


Figure 12: Two processes send Ping and Pong messages to each other over an IP network.

3.2 Event interception

Event interception is a fundamental pattern supported in Kompics. It allows you to extend the functionality of a system without changing it. For example, look at the architecture in Figure 6. Assume that you start only with Component1 which processes MessageA events. Without making any changes to Component1 you can add Component2 to perform some non-functional task, e.g., keep statistics on how many MessageA events were processed.

You can also interpose a component between two components connected by a channel, to perform complex filtering of events, to delay events, or implement some form of admission control for events. Recall the Ping-Pong example in Figure 12. In Figure 13 we interpose a SlowNetwork between MyComponent and MyNetwork. The SlowNetwork delays every message sent by MyComponent by some random delay. In essence we emulate a slower network. (SlowNetwork could be configured to emulate specific fine-grained network conditions.) This allows you to experiment with the (unmodified) Ping-Pong protocol on a network with special properties.

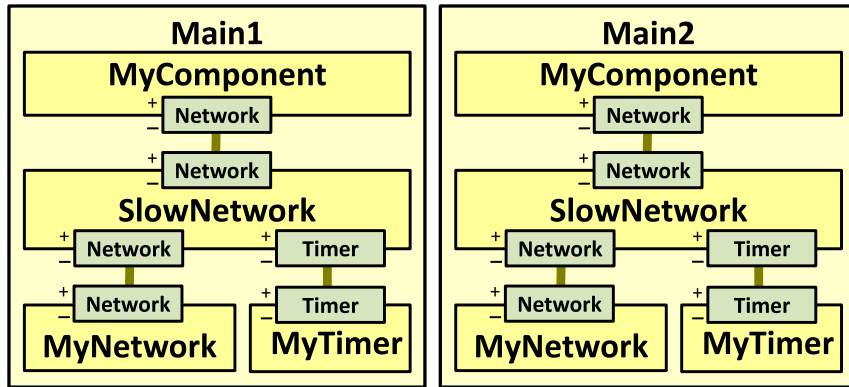


Figure 13: We interposed a SlowNetwork between MyComponent and MyNetwork to emulate network latency.

3.3 Setting and canceling timers

In Kompics alarms and timeouts are provided as a service abstraction through the **Timer** port. If this guideline is followed, different implementations of the **Timer** abstraction can be used for different execution environments. We illustrated the **Timer** port in Figure 1. It accepts two request events, **ScheduleTimeout** and **CancelTimeout**, and it delivers a **Timeout** indication event.

In Figure 14 we illustrate a component that uses a **Timer** abstraction. Typically, when designing a component (e.g., **MyComponent**), one would also design specific timeout event, here **MyTimeout**, as a subtype of the **Timeout** event. Multiple timeout event types can be defined for different timing purposes, so a component can have different event handlers for different timeouts. Code 10 shows how you can schedule a timeout.

```

1 class MyComponent extends ComponentDefinition {
2   Positive<Timer> timer = requires(Timer.class);
3   UUID timeoutId; // ← used for canceling
4   Handler<Start> startHandler = new Handler<Start>() {
5     public void handle(Start event) {
6       long delay = 5000; // milliseconds
7       ScheduleTimeout st = new ScheduleTimeout(delay);
8       st.setTimeoutEvent(new MyTimeout(st));
9       timeoutId = st.getTimeoutId();
10      trigger(st, timer);
11    }
12 }

```

Code 10: *Scheduling a timeout.*

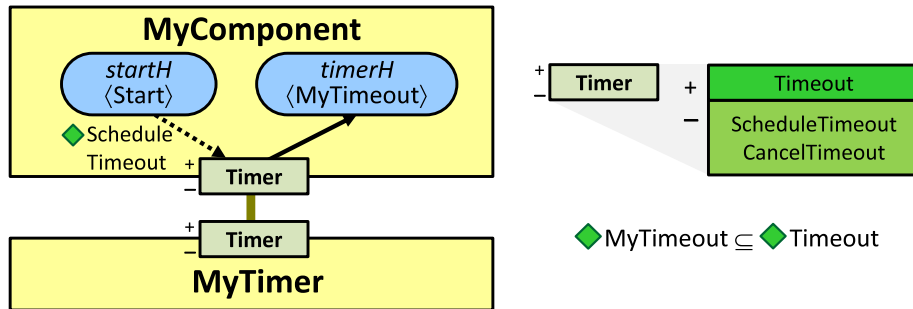


Figure 14: **MyComponent** uses the **Timer** abstraction provided by **MyTimer**.

To cancel a previously scheduled timeout, a component will issue a **CancelTimeout** request on a required **Timer** port. The **CancelTimeout** event needs the unique identifier of the scheduled timeout. Code 11 shows how you cancel a timeout.

```

1 CancelTimeout ct = new CancelTimeout(timeoutId);
2 trigger(ct, timer);

```

Code 11: *Canceling a timeout.*

3.4 Remote service invocation

A common idiom in many distributed systems is sending a request to a remote node and waiting for a response up to a timeout. This entails scheduling a timeout to be handled in case the response never arrives, e.g., if the remote node crashed, and canceling the timeout when the response does arrive. One recommended practice is to send the unique timeout identifier in the request message and echo it in the response message. When the client node gets the response, it knows which timer to cancel. Another recommended practice is to keep a set of all outstanding requests (their timeout ids). This helps with handling either the response or the timeout exclusively: upon handling either the response or the timeout, the request is removed from the outstanding set. Neither the response nor the timeout is handled if the request is not outstanding anymore.

3.5 Higher-level abstractions

In Section 3.1 we showed a point-to-point communication abstraction. In Kompics we have designed and implemented a wide array of higher level fault-tolerant distributed systems abstractions such as: reliable broadcast, failure detectors, leader election, consensus, atomic shared memory registers, replicated state-machines, virtual synchrony, etc. [11]. We show some of these with the case study in Section 5.

4 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, peer-to-peer framework, and case studies presented here, are all available at <http://kompics.sics.se>.

4.1 Java runtime engine and framework

The Kompics execution model admits an implementation with one lightweight thread per component. Since Java has only heavyweight threads, we use a pool of worker threads for executing components. Every component is marked as *idle* (has no events), *ready* (has some events, waiting for worker), or *busy* (executing an event now). Each worker has a dedicated queue of ready components. Workers process one component at a time and one component cannot be processed by multiple workers at the same time. Thus, we guarantee the Kompics execution model whereby handlers of one component instance execute mutually exclusively. The Kompics runtime system supports pluggable component schedulers and allows you to provide your own scheduler. The next subsection highlights the default scheduler.

The **MyNetwork** component embeds the Apache MINA network library and implements automatic connection management and message serialization and Zlib compression.

We are currently investigating a Kompics front-end in Scala. This could immediately leverage the existing Java components and runtime system. Also, it has the potential for more expressive code and for avoiding inversion of control.

4.2 Multi-core component scheduling

Workers may run out of ready components to execute, in which case they engage in *work stealing* [5]. Work stealing involves a *thief*, a worker with no ready components, contacting a *victim*, a 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 queue is a lock-free queue, meaning that the victims and thieves can concurrently consume ready components from the queue.

4.3 Deterministic simulation mode

We provide a special scheduler for system simulation. The system code is executed in deterministic simulation provided it does not create threads.

Next section describes this mechanism in more detail, in the context of the case study simulation architecture.

4.4 Programming in the large

We used Apache Maven to organize the structure and manage the artifacts of the Kompics implementation. 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 true reusability of protocol abstractions and component implementations. You can start a project for a new protocol implementation and just specify what existing abstractions your implementation depends on. They are automatically fetched and made visible in your project. This approach also enables deploy-time composition.

5 Case study

In order to put into perspective the Kompics concepts, patterns, and abstractions, we present a case study where we used Kompics to develop, deploy, stress-test, and simulate a distributed hash-table with atomic consistency. 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 performance evaluation. We show how to specify large-scale experiment scenarios that can be used both for simulation and stress-test interactive execution.

5.1 Peer-to-peer systems architecture

We implemented a component framework with protocols reusable in any peer-to-peer system. Every P2P system needs a bootstrap procedure to assist newly arrived peers in finding a peer which is already in the system and engage in a join protocol. For this, we have a **BootstrapServer** component which maintains a list of online peers for a particular system instance. Every peer embeds a **BootstrapClient** component which provides a **Bootstrap** service to the peer. When a peer starts, it issues a **BootstrapRequest** to the client which retrieves from the server a list of alive peers and delivers a **BootstrapResponse** to the peer. The peer engages in a join protocol with one or more the returned peers 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 peer is still alive. The server evicts peers who stop sending keep-alives.

We also provide a monitoring protocol where a client component on each peer periodically inspects the status of various components of the peers and reports this to a monitoring server. The server aggregates the status of peers, compiles a global view of the system and presents this global view on a web page. The bootstrap and monitoring servers are illustrated in Figure 15, within executable main components.

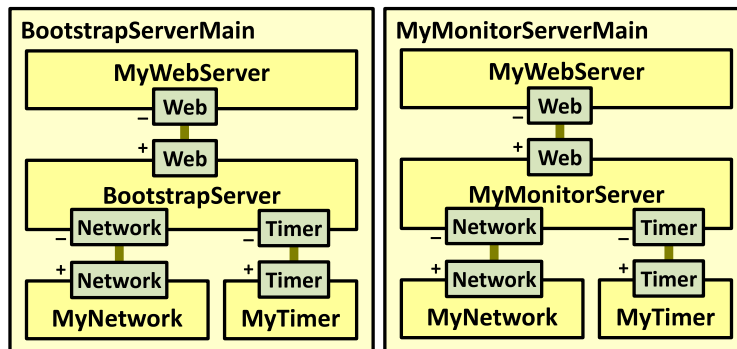


Figure 15: *Generic P2P bootstrap and monitoring servers exposing a user-friendly web interface for troubleshooting.*

We embed the Jetty web server library in the **MyWebServer** component which

embeds every HTTP request into a **WebRequest** event and triggers it on a required a **Web** port. Both servers provide the **Web** abstraction, accepting **WebRequests** and delivering **WebResponses** containing HTML pages with the peer list and global view, respectively.

In Figure 16 we show the component architecture designated for system deployment. Here we have an executable main component which embeds peer, network, timer, web server, and application components. The peer exposes its status through a **Web** abstraction. The HTML page representing the peer's status will typically contain hyperlinks to the neighbor peers and to the bootstrap and monitoring server. This enables users/developers to browse the P2P network over the web, and inspect/troubleshoot the state of each remote peer. The **MyApplication** component may embed a GUI or CLI user interface and issue functional requests (or accept indications) to the **MyPeer** component over the **MyPeerPort**. (**MyApplication**, like the **MyNetwork** or **MyTimer** components may depart from the model guidelines and embed threads, however since it is never executed in simulation mode this poses no problem.)

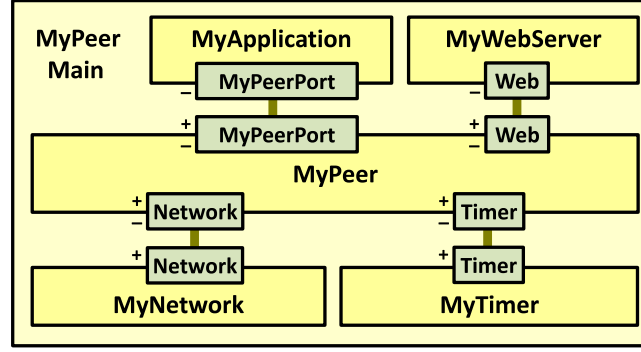


Figure 16: *Component architecture for one peer node. This architecture is designated for system deployment where every peer executes on a different machine and communicates with other peers by sending messages over an IP network.*

The **MyPeer** component is detailed in Figure 17. We have already discussed the bootstrap and monitoring clients earlier. The **PingFailureDetector** component implements an eventually perfect failure detector by sending **Ping** messages to and expecting **Pong** messages from monitored neighbored peers. The **Chord** component uses the **FailureDetector** abstraction and implements the Chord structured overlay network (SON) providing a **SON** abstraction for routing and lookups. **MyReplication** implements a key-based replication scheme on top of **SON** providing a **Replication** abstraction.

This enables us to switch the replication scheme of the system by replacing the **MyReplication** component with a different implementation. The **MyGroupMembership** component implements a group membership abstraction (**GM**) maintaining a consistent view of member peers in each replication group. The **MyDHT** component uses the **SON**, **Replication**, and **GM** abstractions, to implement a distributed hash-table (**DHT**) with atomic consistency, and provides the **DHT** abstraction accepting *put* and *get* operations.

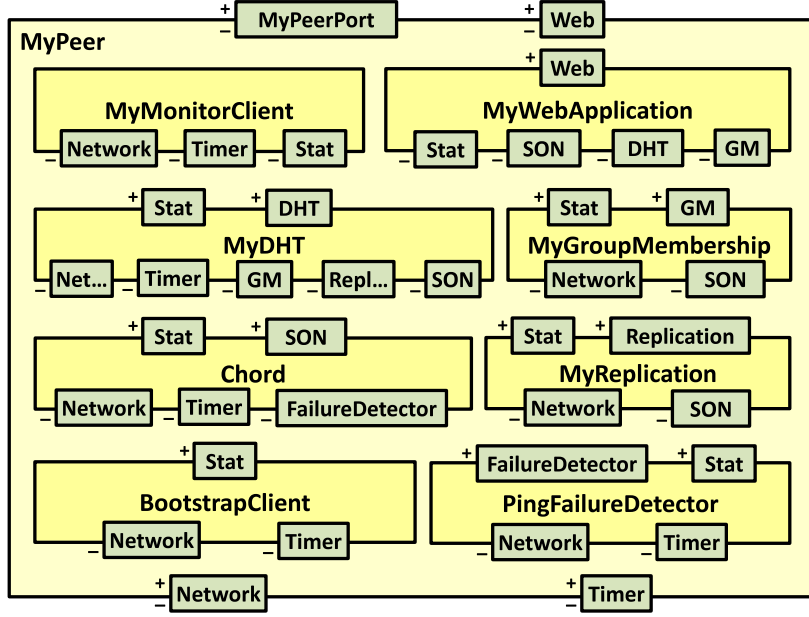


Figure 17: The architecture of the *MyPeer* component. We omit the channels for clarity. In this scope, all provided ports are connected to all required ports of the same type.

Every functional component provides a **Stat** port, accepting **StatusRequests** and delivering **StatusResponses** to **MyMonitorClient** and **MyWebApplication**. **MyWebApplication** provides a web interface to the peer, dumping the status of the peer components and allowing users to issue interactive commands to the **DHT** or **SON**.

We have presented the component architecture of a deployable P2P system. We have actually deployed this system on the PlanetLab testbed and also on our local-area cluster. Using the web interface to interact with the **DHT** (configured with a replication degree of 5) on the local-area network, resulted in end-to-end latencies of $\approx 10\text{-}15\text{ms}$ for *get* operations and $\approx 15\text{-}20\text{ms}$ for *put* operations. This includes the LAN latency (two message round-trips, so 4 times), message serialization (4x), encryption (4x), decryption (4x), deserialization (4x), and Kompics runtime overheads for message dispatching and execution.

We now show how the same system code is executed in simulation mode for stepped debugging or repeatable large-scale performance evaluations. In Figure 18 you see the component architecture for simulation mode. Here, a generic **P2pSimulator** interprets an experiment scenario (described in the next subsection) and issues command events to a system-specific simulator component, **MySimulator**, through a **MyExperiment** port. An issued commands may tell **MySimulator** to create and start a new peer, to stop and destroy an existing peer, or to instruct an existing peer to execute a system-specific operation (through its **MyPeerPort**).

The **P2pSimulator** 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 events and when it runs out of work it passes control to the **P2pSimulator** to advance simulated time.

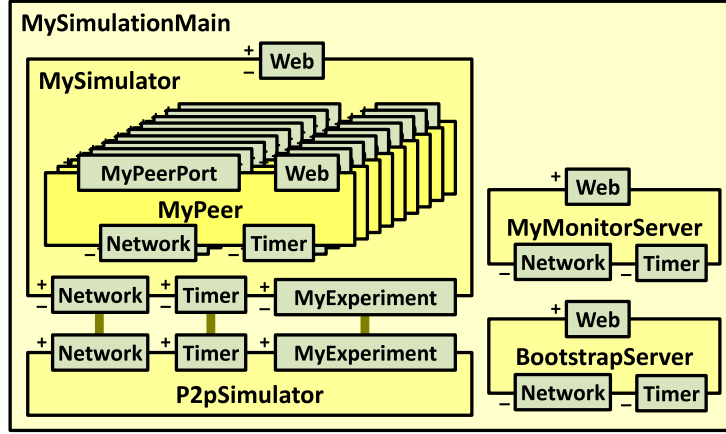


Figure 18: *Component architecture for whole-system simulation. All peers and servers execute within a single OS process in simulated time.*

In simulation mode, the bytecode of the system is instrumented to intercept all calls for the current time and return the simulated time. Therefore, without editing the system code at all, the system can be executed deterministically in simulated time. JRE code for secure random number generators is also instrumented to use the same seed and achieve determinism. Attempts to create threads by the system code are also intercepted and the simulation halts since it cannot guarantee deterministic execution.

Using the same experiment scenario used in simulation, the same system code can be executed in an interactive stress-testing execution mode. In Figure 19 you see the respective component architecture. This is similar to the simulation architecture, however, we use our regular component scheduler and the system executes in real-time, albeit driven from the same experiment scenario.

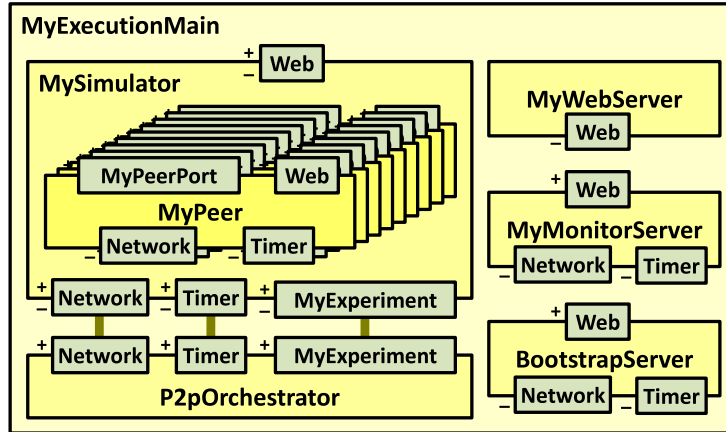


Figure 19: *Component architecture for whole-system interactive stress-test execution. All peers and servers execute within a single OS process in real time.*

The P2pSimulator was replaced with a P2pOrchestrator which provides the Network and Timer abstractions and drives the execution from an experiment scenario.

During development it is recommended to incrementally make small changes

and quickly test their effects. The interactive execution mode helps you with this routine since it enables you to quickly run a small-scale distributed system (without the need for deployment or manual launching of multiple processes) and interact with it using a web browser.

All components whose name does not start with “My...”, as well as `MyNetwork`, `MyTimer`, and `MyWebServer` are generic and completely reusable. All other components in this example are system-specific. They serve as an architectural design pattern and with slight changes they can be adapted for different systems.

5.2 Peer-to-peer system experimentation

We designed a Java domain-specific language (DSL) for expressing experiment scenarios for P2P systems. Such experiment scenarios are interpreted by, e.g., `P2pSimulator` or `P2pOrchestrator`. We now give a brief description of our DSL with a simple example scenario.

A scenario is a parallel and/or sequential composition of *stochastic processes*. We call a stochastic process, a finite random sequence of events, with a specified distribution of inter-arrival times. Code 12 shows an example stochastic process.

```
1 StochasticProcess boot = new StochasticProcess() {{
2   eventInterArrivalTime(exponential(2000)); // ~2s
3   raise(1000, chordJoin, uniform(16));      // 1000 joins
4 }};
```

Code 12: *Example definition of a “stochastic process”.*

This will generate a sequence of 1000 `chordJoin` operations, with an inter-arrival time between two consecutive operations extracted from an exponential distribution with a mean of 2 seconds. The `chordJoin` operation is a system-specific operation with 1 parameter. In this case, the parameter is the Chord identifier of the joining peer, extracted from an uniform distribution of $[0..2^{16}]$. Code 13 shows how the `chordJoin` operation is defined.

```
1 Operation1<Join, BigInteger> chordJoin
2   = new Operation1<Join, BigInteger>() {
3   public Join generate(BigInteger nodeKey) {
4     return new Join(new NumericRingKey(nodeKey));
5   }
6 };
```

Code 13: *Example definition of a system-specific operation taking one parameter.*

It takes 1 `BigInteger` argument (extracted from a distribution) and generates a `Join` event (triggered by the `P2pSimulator` on `MyPeerPort`). In Code 14 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.

In Code 15 we define a process to issues some `Lookup` events.

The `chordLookup` operation takes 2 `BigInteger` parameters, extracted from a (here, uniform) distribution, and generates a `Lookup` event that tells `MySimulator`


```

1 StochasticProcess churn = new StochasticProcess() {{
2   eventInterArrivalTime(exponential(500)); // ~500ms
3   raise(500, chordJoin, uniform(16));      // 500 joins
4   raise(500, chordFail, uniform(16));      // 500 failures
5 }};

```

Code 14: *Example definition of a churn stochastic process.*

```

1 StochasticProcess lookups = new StochasticProcess() {{
2   eventInterArrivalTime(normal(50)); // ~50ms
3   raise(5000, chordLookup, uniform(16), uniform(14));
4 }};

```

Code 15: *Example definition of process that issues lookup operations.*

to issue a lookup for key *key* at the peer with identifier *node*. As you can see in Code 16, a random peer 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> chordLookup
2   = new Operation2<Lookup, BigInteger, BigInteger>() {
3   public Lookup generate(BigInteger node, BigInteger key) {
4     return new Lookup(new NumericRingKey(node),
5                       new NumericRingKey(key));
6   }
7 };

```

Code 16: *Definition of a lookup operation taking two parameters.*

We defined three stochastic processes: *boot*, *churn*, and *lookups*. Code 17 shows how we can compose them.

```

1 boot.start(); // start
2 churn.startAfterTerminationOf(2000, boot); // sequential
3 lookups.startAfterStartOf(3000, churn); // in parallel
4 terminateAfterTerminationOf(1000, lookups); // terminate

```

Code 17: *Sequential and parallel composition of stochastic processes.*

The experiment scenario starts with the *boot* process. 2 seconds (simulated time) after this process terminates, starts the *churn* process. 3 seconds after *churn* starts, the *lookups* process starts, now working in parallel with *churn*. The experiment terminates 1 second after all lookups are done.

Putting it all together, Code 18 shows how you define and execute an experiment scenario using our Java DSL:

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). 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

```

1 class MySimulationExperiment {
2     // system-specific operations definitions
3     static Scenario scenario1 = new Scenario() {
4         StochasticProcess boot = ... // see above
5         StochasticProcess churn = ...
6         StochasticProcess lookups = ...
7         boot.start()
8         churn.start...
9         lookups.start...
10        terminate...
11    }
12    public static void main(String[] args) {
13        scenario1.setSeed(rngSeed);
14        scenario1.simulate(MySimulationMain.class); // ← simulation
15        // scenario1.execute(MyExecutionMain.class); // ← local execution
16    }
17 }

```

Code 18: *Using a scenario to drive a simulation or a real-execution experiment.*

to interact with and monitor the system over the web during while the experiment is running.

We 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 mode, driven from the same experiment scenario.

We ran simulations of this system and we were able to simulate a system of 16384 peers in a in 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 you can see in Table 1.

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

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

6 Related work

Kompics is related to work in several areas, including: concurrent programming

models [4, 23, 25], reconfigurable component models for distributed systems [6, 8, 24], reconfigurable software architectures [17, 20, 9, 2], and event-based frameworks for building distributed systems [14, 15, 22, 26].

Kompics’s message-passing concurrency model is similar to the actor model [1], of which Erlang [4], the Unix process and pipe model, Kilim [25] and Scala [23] 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 is also found in other message-passing systems, such as Singularity [10]. 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 (ports may be connected to potentially many other components).

Kompics’s execution model is similar to the latest version of Erlang which is based on separate run queues per scheduler [18] (similar to Kompics’s multiple workers with private work queues). Erlang’s schedulers load balancing algorithm differs from Kompics, where a combination of work migration and stealing is used. In Erlang, the system measures the total number of runnable processes four times per second, and schedulers compare their local workload with the average global workload then take a local decision on whether to migrate their work to a different scheduler. Workers with no work will try and steal work. In Kompics, we adopt a batched work stealing approach. Scala supports concurrent schedulers that take work from a shared global work queue [12], which becomes a bottleneck in multi-core systems. Kilim’s execution model, in contrast, is based on active actors and Kilim’s lightweight thread model, where an actor is executed by its own lightweight thread [25].

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 [2], OpenORB [8], Fractal [6] and LiveObjects [24]. However, with the exception of LiveObjects, these component models are inherently client-server models, with RPC interfaces. In software architectures, such as ArchJava [2], ports are used to manage explicit dependencies between components. LiveObjects has the most similar goals to Kompics of supporting encapsulation and composition of distributed protocols using components. 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. In LiveObjects, composite components as a special type of component.

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 [19], such as OpenORB [8], or dynamic software architecture models, such as Fractal [6], Rapide [17], and ArchStudio4/C2 [9]. 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. This is similar to the ArchStudio4/C2 approach.

Kompics support for event-based programming to compose protocol layers is similar to Wids [16] and Mace [14], both of which are frameworks for building P2P systems. Appia [22] is another related protocol composition framework. These systems support distributed messaging using events that encapsulate routing information and this contrasts with the alternative for distributed objects that establish sessions between network endpoints, as in CORBA [13] and WS-Session for web services [7]. However, none of these frameworks provide general component or concurrency models.

Other related work includes the Pi-Calculus [21] that uses names and co-names for actions, similar to our notion of the polarity of ports. In terms of programming, our event-based programming model results in an *inversion of control* programming style, similar to programming for graphical user interface frameworks, such as Java Swing.

7 Conclusions and future work

We are witnessing a boom in distributed services and applications. Many companies independently develop complex distributed systems from the ground up. The current situation is comparable to the times when companies were independently developing different networking architectures before ISO/OSI model [27] came along. We believe that industry would benefit tremendously from the availability of a systematic approach to building distributed systems.

We hope this work will open up the path to more systematic research on dynamically evolving distributed system architectures, currently an under-explored area.

We are currently working on a more expressive programming model to enable event-driven programming without inversion of control. Future work includes distributed component deployment and dependency management to enable autonomously evolving distributed systems and transactional reconfiguration of the component architecture. We plan to extend our simulation environment with a stochastic model checker to improve the testing and debugging of Kompics-based systems. A particular implementation of a distributed abstraction is replaced with an implementation that generates random admissible executions for the replaced abstraction. This enables the stochastic verification of the overlying abstractions which are subject to various legal behaviors.

We have successfully used the Kompics component model as a teaching tool in two courses on distributed systems. Kompics enabled students both to compose various distributed abstractions and to experiment with large-scale overlays and content-distribution networks both in simulation and real execution. Students were able both to deliver running implementations of complex distributed systems and

to gain insights into the dynamics of those systems. We believe that making distributed systems easier to program and experiment with, will significantly improve the education process in this field and will lead to better equipped practitioners.

References

- [1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
- [2] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *European Conference on Object Oriented Programming ECOOP 2002*, pages 185–193, 2002.
- [3] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD dissertation, The Royal Institute of Technology, Sweden, 2003.
- [4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.
- [5] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.
- [6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. 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] Wu Chou, Li Li, and Feng Liu. Web service enablement of communication services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 393–400, Washington, DC, USA, 2005. IEEE Computer Society.
- [8] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, February 2008.
- [9] Eric M. Dashofy, Hazeline U. Asuncion, Scott A. Hendrickson, Girish Suryanarayana, John C. Georgas, and Richard N. Taylor. Archstudio 4: An architecture-based meta-modeling environment. In *ICSE Companion*, pages 67–68, 2007.
- [10] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.
- [11] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.
- [12] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *JMLC*, pages 4–22, 2006.
- [13] Michi Henning and Steve Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [14] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. *SIGPLAN Not.*, 42(6):179–188, June 2007.
- [15] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.
- [16] Shiding Lin, Aimin Pan, Rui Guo, and Zheng Zhang. Simulating large-scale p2p systems with the wids toolkit. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 415–424, Washington, DC, USA, 2005. IEEE Computer Society.

- [17] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, 1995.
- [18] Kenneth Lundin. Invited talk: The future of erlang. In *7th ACM SIGPLAN workshop on ERLANG*, 2008.
- [19] Pattie Maes. Computational reflection. In *GWAI '87: Proceedings of the 11th German Workshop on Artificial Intelligence*, pages 251–265, London, UK, 1987. Springer-Verlag.
- [20] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.
- [21] Robin Milner. *Communicating and mobile systems: the π -calculus*. Cambridge University Press, New York, NY, USA, 1999.
- [22] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS '01*, pages 707–710, Washington, DC, USA, 2001. IEEE Computer Society.
- [23] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05*, pages 41–57, New York, NY, USA, 2005.
- [24] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with live distributed objects. In *European Conference on Object Oriented Programming ECOOP 2008*, pages 463–489, 2008.
- [25] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object Oriented Programming*, 2008.
- [26] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, volume 35, pages 230–243, New York, NY, USA, December 2001. ACM Press.
- [27] Hubert Zimmermann. The iso reference model for open systems interconnection. *IEEE Transactions on Communications*, 28:425–432, April 1980.