

Master's Thesis

Evaluation and Extension of an Implementation of Flow-Based Programming

in partial fulfillment of the requirements for the degree
Master of Science Informatik
submitted to
Fachbereich Mathematik, Naturwissenschaften und Informatik of the Fachhochschule
Gießen-Friedberg

Sven Steinseifer
July 2009

Advisor: Prof. Dr. Thomas Letschert
Co-advisor: Prof. Dr. Hellwig Geisse

To Jesus Christ, my lord and savior

Contents

1	Introduction	1
1.1	PARASUITE	1
1.1.1	Technology	1
1.2	Goals	2
2	What Is Flow-Based Programming?	3
2.1	History	3
2.2	FBP – a Programming Paradigm	3
2.2.1	The FBP Language	3
2.2.2	Hierarchical Decomposition	5
2.2.3	Data Grouping	6
2.3	Component Model	6
2.3.1	Components and Processes	6
2.3.2	Asynchronous Message Passing	6
2.3.3	Scheduling	7
2.3.4	Subnets and Substreams	8
2.4	Properties of FBP	8
2.4.1	Loose Coupling	8
2.4.2	Efficiency	9
2.4.3	Simplicity	9
2.4.4	Problems and Challenges	10
3	Similar Concepts	12
3.1	Kahn Process Networks	12
3.2	Pipes and Filters	13
3.3	Active Object	14
4	Improving FBP	15
4.1	Introducing a Port-Selection Mechanism	15
4.2	Variable or Unlimited Buffer Sizes	15
5	Java Implementation of FBP	17
5.1	Usage of the JavaFBP Framework	17
5.1.1	The FBP Language of JavaFBP	17
5.1.2	Implementing Components	17
5.1.3	Creating Subnets	17
5.2	How does JavaFBP implement FBP?	18
5.2.1	Statical Design – Important Classes	18
5.2.2	Dynamics – Which Class Does What?	19
5.2.3	Flaws, Inconveniences and Fixes	20
6	Scalability of JavaFBP	24
6.1	The Testing Environment	24
6.2	Network Topologies	25
6.2.1	The Serial Topology	25
6.2.2	The Parallel Topology	26
6.2.3	The Triangle Topology	27

6.2.4	Rationale	29
6.2.5	Test Program	29
6.3	How many processes does the JRE allow?	30
6.4	How does JavaFBP scale?	32
6.4.1	Statistically Rigorous Performance Evaluation	32
6.4.2	Tools	33
6.4.3	Serial Networks	33
6.4.4	Parallel Networks	38
6.4.5	Triangle Networks	43
6.4.6	Conclusions	46
6.5	Ideas For Improvement	46
6.5.1	Increasing Memory, Distribution	46
6.5.2	Cooperative M:N Scheduling	47
7	Conclusion	49
7.1	Open Issues and Further Research	49
A	Source Code	IV
A.1	JavaStats	IV
A.1.1	javastats.conf	IV
A.1.2	performance.py	IV
A.2	Test Program	VI
A.2.1	Class run.Main	VI
A.2.2	Class networks.TestNetwork	VIII
A.2.3	Class networks.SerialNetwork	VIII
A.2.4	Class networks.ParallelNetwork	IX
A.2.5	Class networks.TriangleNetwork	X
A.2.6	Class datastructures.Triangle	XI
A.2.7	Class components.Generate	XI
A.2.8	Class components.SierpinskiSelector	XII
A.2.9	Class components.Square	XIII
A.2.10	Class components.TriangleGenerator	XIII

1 Introduction

Today, there are some software applications available which provide an environment where the end user can create new applications or modify existing ones by herself or himself. Lieberman et al. [2006] call this idea to let the user develop software “end-user development” (EUD). EUD is provided in very different flavors. Some environments provide scripting languages to allow their behavior be customized to the users’ needs. These languages are very close to or are in fact traditional programming languages. With domain-specific languages often a declarative approach is used.

Widespread tools for EUD are spreadsheet applications like Microsoft Excel. In such applications the user specifies computations by entering formulas in table cells. While spreadsheet applications often only allow a very limited amount of data to be handled, there are some special applications in the area of data mining like KNIME¹ which can be programmed by the end user to do computations on large sets of data. This is done by allowing the user to graphically connect some predefined components which do some computations on incoming data and produce some output.

1.1 PARASUITE

PARASUITE is a software for aiding decisions in several life-cycle phases of industrial goods and plants. For this purpose, data collected from sensors in these plants or maintenance reports together with general product data is stored in a database. PARASUITE offers some facilities to view this data, to create summaries, reports and analyses of it. Because the producers and operators of the plants know much better how to interpret the data, a facility is needed which allows the users of PARASUITE to specify the algorithms for analyzing the data.

The developers of PARASUITE settled on a concept similar to that of KNIME. The user can select from a set of predefined, parameterizable components and create a directed graph with instances of the components as vertexes and connections between them as edges. The data flows from data-source components through calculation components to data sink components.

The concrete concept used is called “Flow-Based Programming” (FBP) and will be presented in detail in chapter 2.

1.1.1 Technology

PARASUITE is a client-server application which is developed using the Java programming language. The client is based on the Eclipse Rich Client Platform² while the server runs on top of a JBoss Application Server³ connected to a MySQL⁴ database which contains the data. The framework which makes FBP programs run is called JavaFBP and is covered in detail in chapter 5 of this thesis. While FBP programs are created and modified mostly in the client software, they will be executed on the server for performance reasons.

¹Konstanz Information Miner, www.knime.org

²http://wiki.eclipse.org/index.php/Rich_Client_Platform

³<http://www.jboss.org/jbossas/>

⁴<http://www.mysql.com/>

1.2 Goals

First of all, the programming paradigm, FBP, shall be introduced in chapter 2 along with advantages, disadvantages and challenges resulting from employing it. Furthermore, FBP shall be compared to similar concepts (chapter 3). Chapter 4 tries to identify some issues with the FBP concept which might be improved – especially with regard to end users having to deal with them.

Chapters 5 and 6 constitute the core of this thesis. Chapter 5 presents a Java implementation of the FBP concepts called “JavaFBP”. Some issues regarding the functionality of JavaFBP are pointed out, and it is shown how the author designed and implemented solutions to them. Before summing up the results in chapter 7, it is examined in chapter 6 whether there are some restrictions regarding the size of FBP networks in the current implementation and how JavaFBP scales with increasing sizes of the networks. Additionally, some solutions are proposed to overcome the restrictions.

2 What Is Flow-Based Programming?

2.1 History

“Flow-Based Programming” is a programming paradigm developed by John Paul Morrison at IBM Canada in 1969 and 1970 [Morrison, section “General”]. He states that it was based on the idea of coroutines [Morrison, 1994, p. 15]. Instead of patenting the concepts, they were put into public domain in 1971 (ibid., p. 14). Until now there have been multiple implementations of FBP (in chronological order, Morrison, “FlowBasedProgramming”, section Implementations):

- Advanced Modular Processing System (AMPS) – implementation in IBM S/360, S/370 and S/390 Assembler
- Data Flow Development Manager (DFDM) – implementation in some Assembler and PL/1
- Threads – implementation in C
- JavaFBP – implementation in Java
- C#FBP – implementation in C#

The following description of FBP will be based on Morrison’s book [Morrison, 1994]. But some details may differ from what is described there because they are implemented differently in the current implementations.

2.2 FBP – a Programming Paradigm

FBP is a programming paradigm. The programmer connects instances of predefined software components. In FBP such instances are called “processes”. Each component has a set of ports. These are the interfaces of the component and connections are created between them. There are essentially two types of ports: input ports and output ports. Components receive data via their input ports and provide data at their output ports. Some components do not have input ports, some do not have output ports. The former can be considered data sources and the latter are data sinks. Components having both types of ports typically perform some transformation on their input data and send the results to their output ports.

2.2.1 The FBP Language

FBP does not specify a concrete syntax. But every language implementing FBP contains (at least) three primitives:

- process declarations,
- connection declarations and
- configuration statements.

2 What Is Flow-Based Programming?

Process declarations specify which processes are used in the FBP program. They contain the type of the process (i.e., the component) and a unique name so that the processes can be distinguished from each other.

Connection declarations specify how the processes of the program are connected. They typically contain the name of the sending process, the source port, the name of the receiving process and the destination port. The processes' names are those given in the process declarations. The ports are those specified by the components. A source port always belongs to the sending process and a destination port belongs to the receiving process.

Configuration statements are used to configure the processes. Not all processes need configuration, but some need to be parameterized. Imagine a process reading data sets from a file. We do not want to hard code the file name in the component because we then would have to modify it for each separate file. Instead, we would like to specify the file name when we use the component. In FBP there is no separate mechanism for providing configuration parameters to processes. The component designer simply specifies one or more input ports at which the component expects configuration parameters. Configuration statements are used to provide this values. They contain the parameter value (or values) which should be used, the name of the process to be configured and the input port receiving the parameter values.

As stated above, FBP does not specify a concrete syntax for these primitives. DFDM came with a special language, JavaFBP implements these in form of Java methods. Processes and connections may be considered nodes and edges of a directed graph. So the language could also be of a graphical nature rather than a textual one¹.

There are some rules how these FBP primitives have to be applied. An output port can only be connected to one input port. It is not allowed to connect it to two or more input ports (figure 2.1). Unless the output port is declared “optional” in the component’s specification, it has to be connected. It is allowed, however, to connect multiple output ports to a single input port (figure 2.2) or leaving it unconnected.

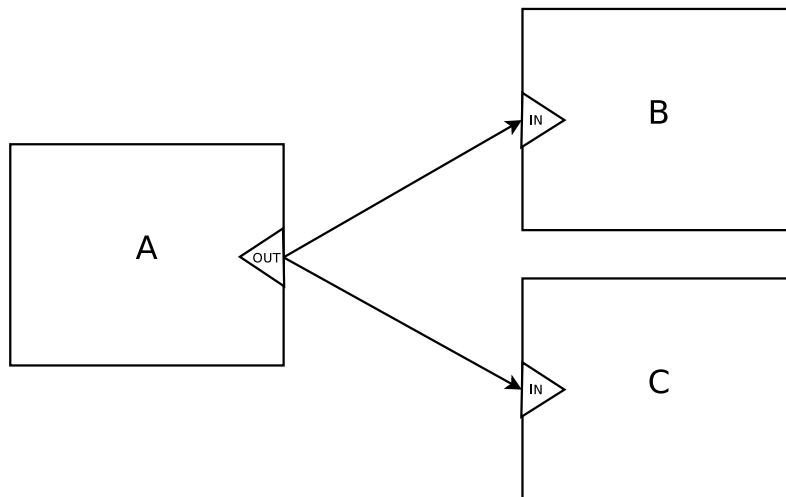


Figure 2.1: one-to-many connections are forbidden

The result of connecting processes using these primitives may be called a “data-flow network”, because the data “flows” from the data sources through the immediate processes to the data sinks.

¹In PARASUITE we have three languages: a graphical one – currently under development. This will be transformed by the client software to an XML-based language, which in turn will be interpreted and translated to the Java-based language of JavaFBP which is described in section 5.1.1

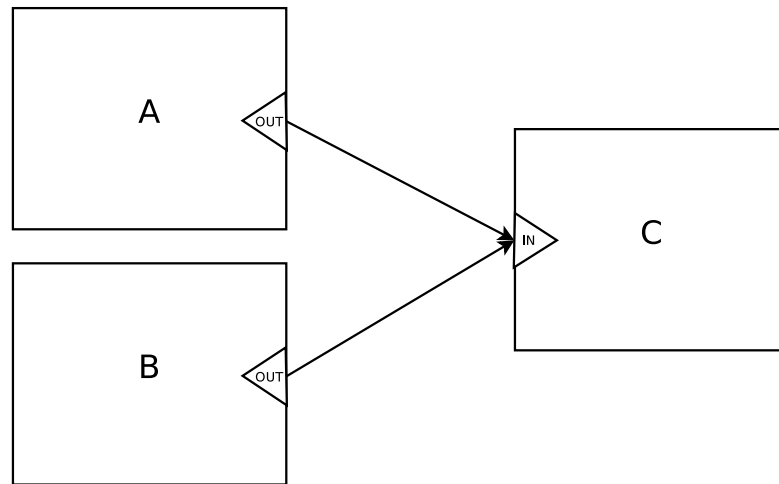


Figure 2.2: many-to-one connections are allowed

2.2.2 Hierarchical Decomposition

It is possible to use hierarchical decomposition to build data-flow networks. That means, one can build a component out of several components. Such a component is called a “subnet” in FBP terminology. Ports of the subnet will be assigned to ports of the inner components. Such a subnet can be used like a normal component in a network. The user of the subnet does not need to know he is using a subnet rather than a normal component.

Have a look at figure 2.3. Process e appears like a normal process in the network but is of type E which is a subnet composed of two instances of C and one instance of G . Also note that the process names in the subnet ($c1$ and $c2$) do not interfere with those in the main network. Although their names and type are equal, they are different instances of C .

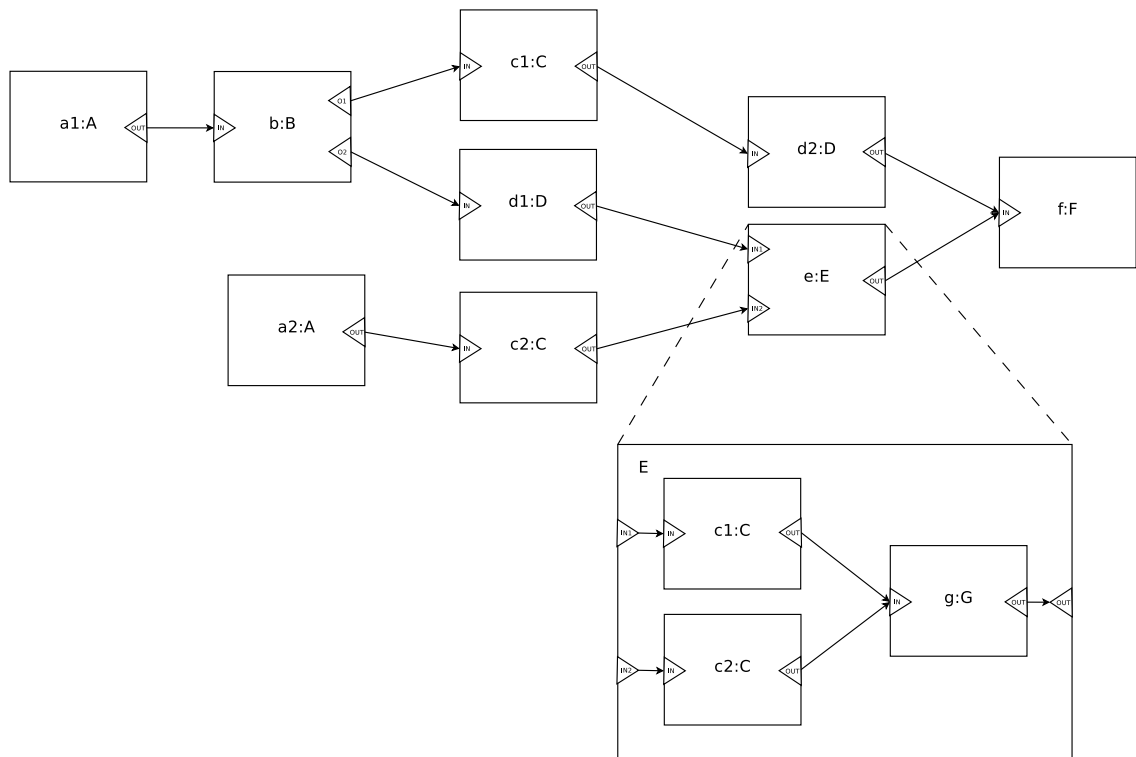


Figure 2.3: FBP network using an instance of a subnet component

2.2.3 Data Grouping

Another facility of FBP is that one may have grouped data. That means that data can be grouped according to certain criteria and processed separately. This is often used in conjunction with components performing aggregate functions like summation or calculating an average value. With this facility, it is possible that a process performs its operation for each group separately. Some components may not support grouped data. If this functionality is needed, they can be embedded in a so-called “substream-sensitive” subnet and the subnet will restart the embedded components for each group.

2.3 Component Model

The previous section gave a superficial overview how an FBP program might look like. But some terms like “component”, “connection” or “data” were simply used without defining them. In this section, this gap shall be filled.

2.3.1 Components and Processes

FBP processes are very similar to operating system processes. They do not get called or call other processes but get started by a so-called “driver” and may run concurrently. Furthermore, they do not share memory but pass messages instead. As stated in section 2.2, processes are instances of components. This means, components are similarly related to processes as classes are related to objects in object-oriented programming. Components can be considered templates for creating processes. They specify properties and behavior of processes. In FBP, properties most notably include number and type of ports belonging to the process. The component behavior is the code which should be executed by the process.

The driver is the interpreter of FBP programs. It also provides the application programming interface (API) which is used by the component implementations, and it is responsible for scheduling the FBP processes.

Components may be implemented in different programming languages if the API is available in the concrete language. They can even be distributed across different computers and be hosted on distinct platforms (operating system, hardware).

2.3.2 Asynchronous Message Passing

Processes pass messages asynchronously via named ports. Typically, the API provides statements for receiving messages from input ports and sending messages to output ports. “Asynchronously” means that the sender does not block until the receiver issues a receive statement. Instead, a bounded buffer is provided at each input port which queues the incoming messages in first-in-first-out (FIFO) order (see figure 2.4). Because the buffer is bounded, the sender does block when the buffer is full and has to wait for the receiver to do a receive. When the buffer is empty, the receiver blocks. Otherwise, data is available, and the receiver can go on processing without blocking. Furthermore, processes can close their output ports to tell the receiving process that no more data is available. A blocking receive statement then returns and indicates end of data. When a process terminates, all output ports will automatically be closed so that upstream processes do not keep waiting for data.

A message is called “Information Packet” (IP) in FBP. Each IP is a container for a piece of information or a data chunk. There is no prescription regarding the granularity of the data chunks or allowed data types. Putting all data in a single IP results in a strictly sequential schedule of serially connected processes. The smaller the IPs are, in relation to the overall data, the more parallel the processes can be scheduled. On the other hand, small IPs increase the communication overhead.

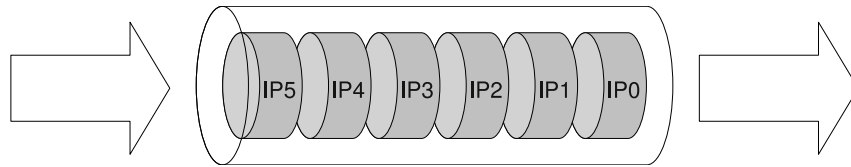


Figure 2.4: a FIFO queue

Another feature of IPs is that they cannot be lost. As soon as an IP enters a process or is created, it is owned by this process. The process can do two things with this IP: pass it on through an output port or dispose of it. This is called “dropping”. Dropping must be done explicitly via a special statement. When the process deactivates (see 2.3.3), it is checked whether it still owns IPs or not. If the former is the case, depending on the implementation, a warning or an error will be raised.

In section 2.2.1 configuration statements were mentioned as a means of parameterizing processes. These statements referred to ports of processes at which configuration data is expected. They create a single IP containing the parameters and bind it to the specified port. Such an IP is called “initial information packet” (IIP). The component will receive it once at this port, and afterwards the port is closed.

If two or more output ports are connected to a single input port, it is not determined how the streams of IPs will be merged. But it is guaranteed that the IPs of each stream will come in in FIFO order.

2.3.3 Scheduling

There are certain rules according to which processes in FBP get scheduled – independent of the underlying implementation of concurrency. A process can be in one of the following states at a time:

- not yet initiated
- active
- suspended on send
- suspended on receive
- inactive
- terminated

Figure 2.5 contains a state chart which describes the behavior.

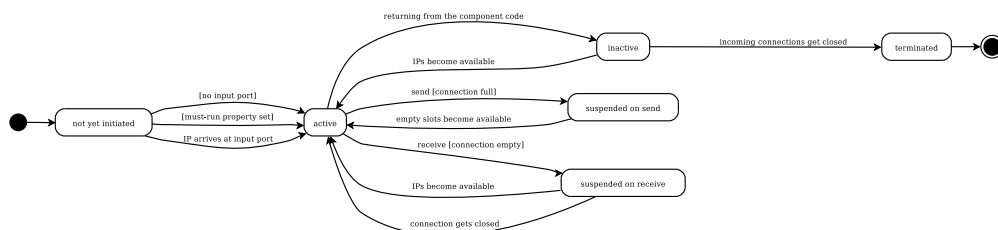


Figure 2.5: state chart – states of a process

All processes of a FBP network start in the “not yet initiated” state. As soon as data arrives at any input port of a process (ports with IIPs do not count), its state changes to “active”. This means, that this process is ready to run. Its entry point will get called eventually by the driver. Some processes do not have input ports. Their state will change to “active” immediately.

When the connection at an output port is full, a send statement referring to this port has to block. This is achieved by putting the process into the “suspended on send” state. The process remains in this state until there is some space again in the connection’s buffer. Then the state is set to “active” and the process may continue to execute.

In almost the same manner, a process will change to “suspended on receive” if it does a receive on an empty connection. As soon as an IP arrives, its state will be set to “active” again. A process cannot check whether IPs are available or not without doing a receive. There also is no facility to wait for IPs arriving at any input port and receive from the port which has IPs available.

If a process leaves its component’s code (e.g. by a return statement), it may enter either the “terminated” or “inactive” state. The former happens when all its input ports are closed. This is the final state of a process. In the “inactive” state it may possible that some IPs arrive at the input ports. In this case, the process is activated again. If a process is “inactive” and all its input ports get closed, it will terminate.

It may happen that a process never gets activated. This is the case when no IP arrives at its input ports. In some cases, it is desirable that this process still gets activated once. A prominent example for this is a component which counts its incoming IPs and provides the count. This component would normally output nothing if no IP arrives, but it should output zero in this case. Components which require that their instances get activated at least once can be specially configured so that the scheduler activates them even if no IP arrives at their input ports (“must-run” attribute).

2.3.4 Subnets and Substreams

The data grouping facility mentioned in section 2.2.3 is realized using so-called “bracket IPs”. Each group is a stream of consecutive IPs enclosed in such bracket IPs. An “open” bracket starts a new substream and a “close” bracket marks the end of the substream. It is possible to nest substreams by nesting pairs of open and close brackets.

Some components may be written to react on bracket IPs. But accordingly configured subnets may show a special behavior when bracket IPs are encountered on the subnet’s input ports: Input ports of subnets may be configured “substream sensitive”. This means, they will remove the outer brackets and close down the connection on encountering “close” brackets. This means, that the inner processes of the subnet will terminate. Normally, terminated processes cannot be activated again, but subnets have the ability to restart the inner processes. So this will happen when a new substream starts. This has the effect that each substream will be processed separately by the subnet. Output ports of the subnet may be configured “substream sensitive”, too. These will put back the bracket IPs which were removed at the input ports to keep the grouping intact.

2.4 Properties of FBP

2.4.1 Loose Coupling

FBP promotes loose coupling between components. There are some prerequisites which help promoting loose coupling. The first one is the sole use of message passing for communication between components. Coupling which results from using shared memory between components – called “common-environment coupling” in Yourdon and Constantine [1979] – is ruled out because there is conceptually no shared memory in FBP. Furthermore, there is only data passed between components, not control (e.g. in form of function or method calls). Yourdon and Constantine call this “data coupling” and consider this the lowest coupling in regard to information flow between components (ibid., pp. 90, 91).

A second prerequisite is the provision of a standard interface between every component of the system, namely ports providing or accepting information packets. The “complexity of the interface” (Yourdon and Constantine) and thus the resulting coupling only depends on the used data structure and not on how the data is passed (*ibid.*, p. 89, 90). But choosing a too general data structure may require introducing logic for parsing and formatting in each component. This can mean that more processing time is used for parsing and formatting than doing the actual task. Thus, the chosen data structure may influence the performance and internal complexity of the components [Shaw and Garlan, 1996, p. 22].

Another important prerequisite is that the components do not know each other. There is no direct reference to another component inside of a component. Hence, the binding of components may be deferred until execution time. According to Yourdon and Constantine this leads to the lowest possible coupling in regard to the “binding time of intermodular connections” (*ibid.*, pp. 93-98). In fact, in PARASUITE it is possible and even intended to configure FBP networks at runtime.

The promotion of loose coupling has several positive effects: It eases testability because each component can be fully tested on its own. Just provide some input data and have a look at the output whether it matches the expectation. Second, it promotes reuse (Buschmann et al. 1996, p. 68; Shaw and Garlan 1996, p. 22). Third, it eases maintainability: “. . . old filters can be replaced by improved ones” [Shaw and Garlan, 1996, p. 22, Shaw and Garlan call components “filters”]. Finally, adaptability is fostered: Components may be arbitrarily rearranged [Buschmann et al., 1996, pp. 62, 68] and applications can be created by rapid prototyping (*ibid.*, p. 70).

2.4.2 Efficiency

FBP can be considered an instance of a “pipes-and-filters” system as appearing in some literature (e.g. Shaw and Garlan 1996 and Buschmann et al. 1996) with processes as filters and connections as pipes (see section 3.2). According to Shaw and Garlan [1996] such systems “naturally support concurrent execution” (*ibid.*, p. 22). Especially on multi-processor machines they may be executed more efficiently than strictly sequential code [Buschmann et al., 1996, p. 68]. But there might be some hindrances to this (according to *ibid.*, p. 69):

- costs of data transmission,
- components reading all incoming data before producing output (e.g. components that perform sorting),
- context switches and
- overhead for synchronizing components if the buffer between them is very small.

2.4.3 Simplicity

As already stated in section 2.2.1 the FBP language and its concepts are quite simple. There might be a graphical representation of the language which enables end users to compose FBP programs themselves. Because components only communicate via ports, they do not affect each other in unexpected ways. Shaw and Garlan [1996] state that the overall input/output behavior of a pipes-and-filters system can be understood “as a simple composition of the behaviors of the individual filters”. At this point, however, should be noted that FBP networks may behave nondeterministically if two or more streams are merged at a single input port (see section 2.3.2).

2.4.4 Problems and Challenges

Deadlocks

Under certain but well-known circumstances a FBP network may deadlock. As Morrison [1994] and Stevens [1991] point out, there are essentially two topologies prone to deadlock. The first is when there is a cycle in the network topology. There are two cases of deadlock:

1. All processes involved in the cycle block on receiving from empty channels. I.e., they expect each respective predecessor to do a **send** (figure 2.6).
2. All processes involved in the cycle block on sending to full channels. I.e., they expect each respective successor to do a **receive** (figure 2.7).

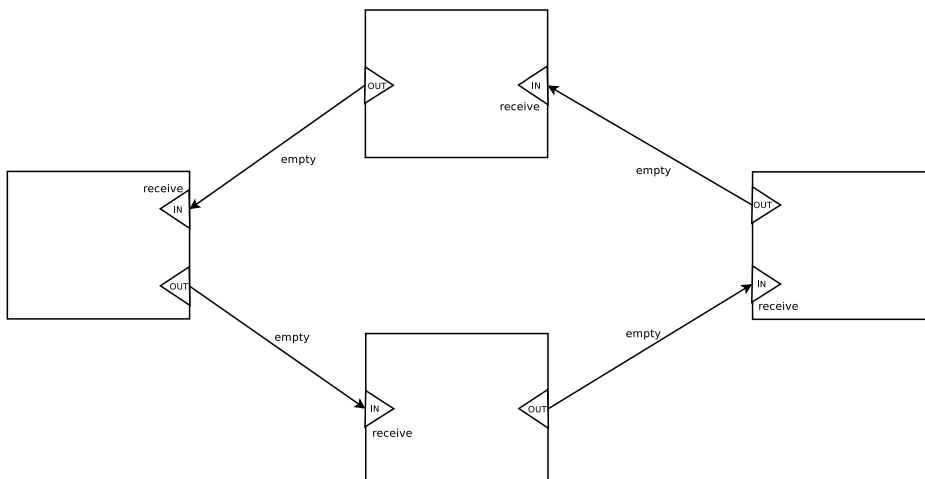


Figure 2.6: processes blocked on receive

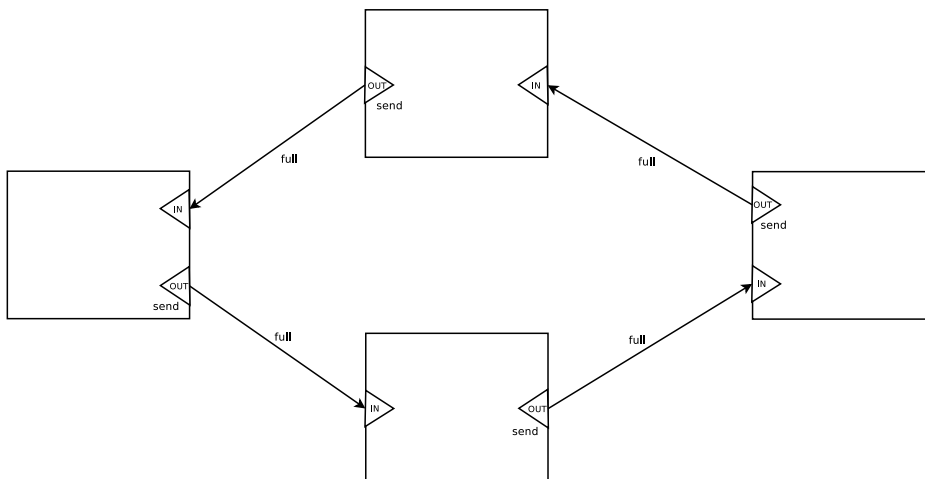


Figure 2.7: processes blocked on send

The second topology prone to deadlock is when two or more streams diverge from a single process and converge at another single process but at different ports (at a single port would be no problem). The deadlock occurs when the sender blocks at sending to a full channel and the receiver blocks at receiving from an empty channel (see figure 2.8).

Deadlocks can be recognized very easily. If there is no process in state “active” and some processes are suspended on **send** or **receive**, there is a deadlock [Stevens, 1991, pp. 222, 223].

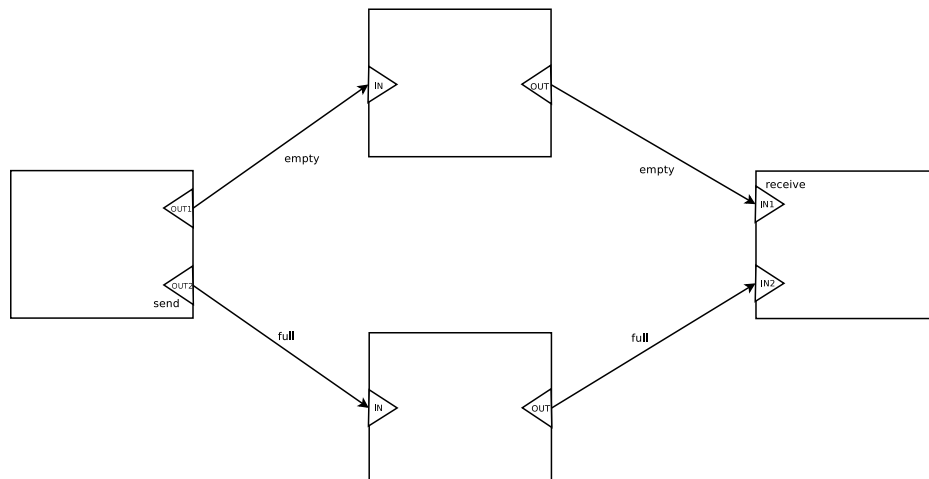


Figure 2.8: processes blocked on diverging streams

Error Handling

It is a challenge to do error handling properly in FBP because there is no shared memory and state.

What can be done easily is reporting errors: An FBP framework might provide a special process which collects error messages from the other processes and write them to a log file or to the console or whatever seems appropriate.

A general and simple procedure would be to stop the whole network when an error occurs in a process. An erroneous process would tell the framework that it cannot go on processing and the framework could terminate the processes because it knows all of them. How such a solution can be implemented is explained in section 5.2.3.

But in some cases it would be nice to do an error recovery and let the network go on running. Unfortunately, error handling often very strongly depends on the use case. What might be a good idea in one case could be completely wrong in another.

3 Similar Concepts

3.1 Kahn Process Networks

A very similar concept to FBP is that of “Kahn Process Networks” (KPN) as described by Kahn and MacQueen [1977]. As with FBP such a process network consists of a set of processes and connections between them. A process may have several input and output ports where it can receive data items from or, respectively, send data items to.

But there are some differences: In a KPN an output port can only be connected to a single input port **and** vice versa in contrast to FBP, where you can connect multiple output ports to a single input port. This restriction leads to a completely deterministic input-output behavior of KPNs because the sequence of data coming from multiple processes does not depend on timing.

With the concept of KPNs there is no separation between the network definition language and the component definition language. It is not intended to create components using different programming languages and connect instances of them in a separate one. Thus, KPNs in their pure form are bound to a single implementation language. This is not to say, that a concrete implementation may not provide support for multiple languages or even distribution across multiple computers.

A facility missing from FBP is the ability to dynamically reconfigure a running network [Kahn and MacQueen, 1977, section 2.4]. In a KPN, a process can decide to replace itself by another one or even by several new ones provided that the number and types of input and output ports remains intact. Or, it can decide to remove itself.

An example for a process replacing itself with two others is given in section 2.5 of *ibid.* which describes an implementation of the sieve of Eratosthenes. The SIFT process in the example is implemented as follows: It has an input port at which it expects a sequence of integer values beginning with two and incremented by one. The prime numbers are provided at an output port. It performs the following steps:

1. Receive an integer value from the input port.
2. Forward the received value to the output port (because it is prime).
3. Replace itself with a new FILTER process connected to a new SIFT process. See figure 3.1 for an illustration.

The FILTER process is configured to filter out values which are multiples of the previously received value.

Another difference between FBP and KPNs lies in the scheduling. The former employs so-called “data-driven” scheduling and the latter employs “demand-driven” scheduling. This means the following: In FBP, processes which are data sources, i.e. have no input ports, are activated first. Processes with input ports are activated as soon as data arrives, hence the name “data-driven”. This means, processes which do not receive data will never get activated (unless they are declared with a must-run attribute, see section 2.3.3).

With KPNs, this is the other way round: Processes which have no output port, i.e. data sinks, are activated first. As soon as they try to receive from an input port, the associated connection is marked “hungry”. This leads to the connected process being activated. The “demand” for data is propagated through the network until a process

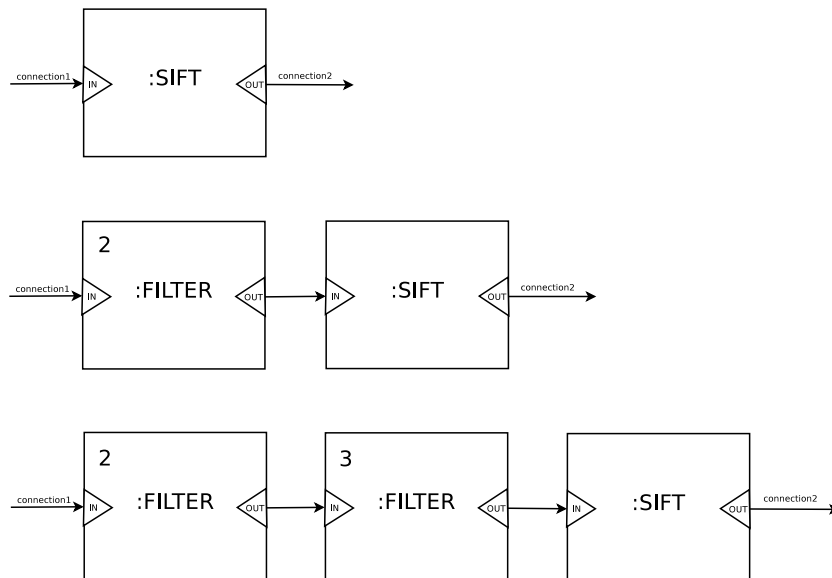


Figure 3.1: a SIFT process replacing itself consecutively with a FILTER and another SIFT process

is activated which can produce data. This means that, in contrast to FBP, a process being a data source might not be activated when there is no demand for its data.

Kahn and MacQueen propose two execution methods for KPNs: The first one is “Coroutine mode of execution” (ibid., section 3.2). In this method, processes are implemented as coroutines. If a process gets activated and does a `receive` on one of its input port, it transfers control to the producing process of this connection. When the latter produces a datum on the connection, control is passed back to the former process. Essentially, this means that the consumer receives the datum directly from the producer without buffering in between. In this mode of execution, the connection capacity is zero. In FBP, the minimum connection capacity is one.

The second method to execute a KPN is called “Parallel mode of execution” (ibid., section 3.3). In this mode, the processes are implemented using real concurrency facilities like operating system processes or threads. This allows to run the processes in parallel instead of interleaved. Connections may be buffered in this mode, allowing producer processes to go on producing some data items until the buffer is full. Kahn and MacQueen call the connection capacity “anticipation coefficient” because the producer anticipates that the consumer will request more data items. In the case the consumer does not do this, the items were produced unavailingly.

Because of the conceptual similarities between FBP and KPNs, the properties of FBP as described in section 2.4 also hold true for KPNs.

3.2 Pipes and Filters

Shaw and Garlan [1996] describe an architectural style called “Pipes and Filters”. An architectural style is a pattern how a software system is structured (ibid., p. 19). Shaw and Garlan “treat an architecture of a specific system as a collection of computational components – or simply *components* – together with a description of the interactions among these components – the *connectors*” (ibid., p.20, their italics). In a Pipes-and-Filters architecture, components have a “set of inputs and a set of outputs” (ibid., p.21). They read data streams from their inputs, possibly do a transformation on them and forward the results to their outputs. Thus, they are called “filters” and the connectors between them are called “pipes” because the data streams flow through them. Furthermore, filters do not share state and do not know each other (ibid., p. 21).

Shaw and Garlan do not go into more detail to accommodate a lot of systems to this style. With that said, FBP can be considered as an instance of a Pipes-and-Filters architecture. In FBP terminology pipes are connections and filters are processes. But KPNs as described in the previous section equally match the description of a Pipes-and-Filters architecture. Thus, “Pipes and Filters” is a generic term for many special concepts.

3.3 Active Object

Lavender and Schmidt [1996] describe a design pattern called “Active Object” in their paper. Essentially, this pattern shows how to implement an asynchronous method call. Caller and callee run in different threads and the invocation of the method should return immediately while the request is processed in the other thread. To shield the client from the complexity, a **Proxy** is provided which contains the available methods. The **Proxy** hands the request to a **Scheduler** which puts it into a queue called **Activation Queue**. This is a bounded buffer containing the not yet handled requests.

In the other thread, the **Scheduler** retrieves requests from the **Activation Queue** and hands them on to a **Servant** which contains the actual logic to be performed.

Results may be passed back using a **Future** object. Such an object is created and returned to the client by the proxy on invocation. Furthermore, a reference to this **Future** object is passed along with the request to the servant. The result of the computation performed by the **Servant** is then put into the **Future** object from which the client may retrieve it.

What has this to do with FBP? – Active Objects are similar to FBP processes. They run in separate threads and contain local state. Proxies are like output ports. They provide an interface to the Active Object with non-blocking methods. The **Activation Queue**, like an FBP connection, contains a bounded buffer which holds the requests.

But there is also a big difference: The **Servant** does not do a **receive** for requests. Instead, it gets called when a request is available. I.e., contrary to what the term “Active Object” suggests, it is rather passive although it is designed to be run in a separate thread.

The Active Object design pattern can be used to implement a concept similar to FBP by treating Active Objects as processes and creating connections via the Proxies. But it allows for more complex interaction by supporting complex interfaces which, on the other hand, can increase the coupling between Active Objects (see section 2.4.1).

4 Improving FBP

4.1 Introducing a Port-Selection Mechanism

As stated in section 2.4.4, a deadlock may occur if two or more streams diverge from a process and converge at another one and the receiver does a `receive` on the wrong port. This could be prevented by providing a mechanism which listens on multiple ports and returns as soon as data on any port is available. A language construct could look like this:

```
portNumber = select(inputPort1, inputPort2, ..., inputPortN)
```

`select` would return the number of the first port on which data is available and the further logic of the component could do a `receive` on the respective port. If all ports have closed down, `select` could return an invalid index like `-1` to signal this.

Morrison argues that such a facility can be replaced by using a single input port for all connections and insert a tagging process between each source and the receiving process. Such a tagging process would insert a tag in each IP which indicates the source of the IP [Morrison, 1994, p. 219] like in figure 4.1. The disadvantage is that this increases the number of processes in a network without reducing the inner complexity of the receiving component because it would have to distinguish between the sources anyway.

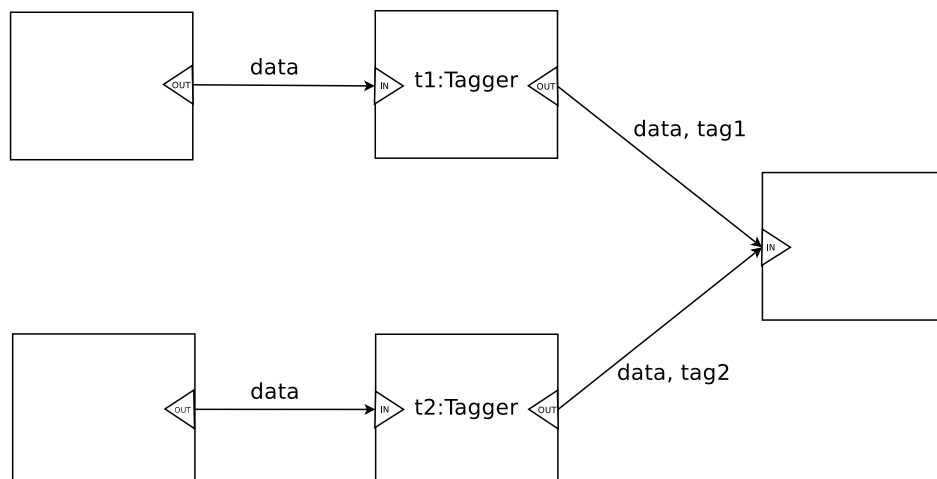


Figure 4.1: Tagger processes inserting tags

The author argues that it is better to increase the complexity of the FBP framework than to force the network developer to deal explicitly with such issues – especially when the network developer is an end user.

4.2 Variable or Unlimited Buffer Sizes

Sometimes a deadlock may be prevented by dynamically increasing the buffer size of connections or providing connections with an unlimited buffer size. Morrison argues that this might simply defer the problem until all memory is consumed. If that happens, it could be that another component in another place tries to allocate some memory and

4 *Improving FBP*

fails to do so and the real error is hidden by the failure of this component (ibid., p. 158).

Deadlock in a FBP network can always be prevented at design time (ibid., p.153). The problem might, again, be to educate the network developer to be able to design networks so that they do not deadlock.

5 Java Implementation of FBP

In this chapter the Java Implementation of FBP called “JavaFBP” shall be examined and evaluated. It is freely available under an open source license from the Sourceforge project site (<http://sourceforge.net/projects/flow-based-pgmng/>).

5.1 Usage of the JavaFBP Framework

To begin with, the usage of the JavaFBP framework shall be presented shortly to introduce the concepts. Readers interested in the details should have a look at the JavaFBP documentation¹.

5.1.1 The FBP Language of JavaFBP

As with all implementations of FBP, JavaFBP provides a language to specify FBP networks. Instead of providing a special syntax, the JavaFBP developers chose to use the Java language. The three language primitives mentioned in section 2.2.1 are provided as methods of the **Network** class.

To create an FBP network, one would extend the **Network** class and implement the **define()** method using the following three methods:

- **component()**: declares processes
- **connect()**: connects ports
- **initialize()**: parameterizes processes

In order to let the Network run, an instance of it must be created and the **go()** method has to be invoked.

5.1.2 Implementing Components

At the time of this writing there only exists an API to implement components using the Java language. A component is created by extending the **Component** class provided by JavaFBP. The port types and names are declared by annotating the class using special Java annotations. To specify the functionality of the component, one has to implement the **execute()** method. For each port, a field has to be declared in the class. Furthermore, their outwardly visible names have to be declared as annotations to the class. Input ports have a **receive()** method to do the **receives** and output ports have a **send()** method to do the **sends**.

IPs are represented by instances of the **Packet** class. They may contain arbitrary Java objects. The **Component** class contains special methods to create (**create()**) and drop (**drop()**) IPs.

5.1.3 Creating Subnets

Subnets can be created by extending the **SubNet** class and implementing its **define()** method as above. In addition to the child processes which should form the subnet, a special process has to be declared for each subnet port. These processes connect the

¹can be found at <http://www.jpaulmorrison.com/fbp/jsyntax.htm>, July 12, 2009

subnet ports with the respective ports of the inner processes. As prescribed by the FBP concept, these subnets can be used like normal components in an FBP network.

5.2 How does JavaFBP implement FBP?

5.2.1 Statical Design – Important Classes

When given the task to create an object-oriented implementation of FBP, a designer likely would come up with the following design – or something similar: A **Network** has a set of **Components** and **Connections** (associations). There are two types of networks: **RootNetworks** and **SubNets**. These are specializations of **Network**. What they have in common is that they both contain components and connections. The difference is that **RootNetworks** get started from the outside and cannot be contained in other **Networks**. On the other hand, **SubNets** will not be stand-alone but will be used as components in a network.

This leads to the next aspect of the design: There are two types of components: **NormalComponents** and **SubNets**. Both may be used in a network, but **SubNets** are implemented by combining instances of other components, while **NormalComponents** are the basic building blocks of FBP networks. This class hierarchies are depicted in the left part of the class diagram in figure 5.1.

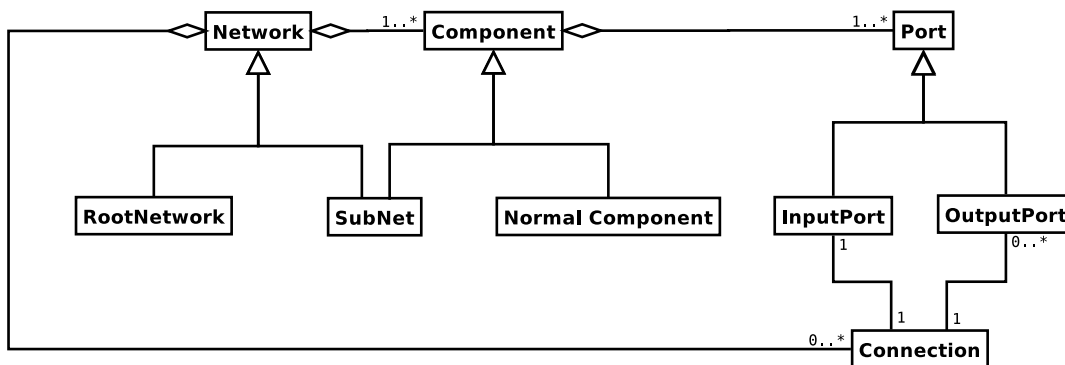


Figure 5.1: class diagram of a hypothetical FBP framework

Components have a set of Ports. A Port is either an input port (**InputPort**) or an output port (**OutputPort**) (specialization). A **Network** has a set of **Connections** (aggregation). Each **Connection** has an input port and multiple output ports (association).² (see the classes on the right in figure 5.1).

The JavaFBP developers followed a similar approach but opted, however, for a single-inheritance class hierarchy (figure 5.2). The **SubNet** class inherits from **Network**, which in turn inherits from **Component**. Normal components extend the **Component** class, subnets extend the **SubNet** class, and root networks extend the **Network** class. For implementing the processes, JavaFBP uses the concurrency facilities of the **Thread** class provided by Java. That is, the **Component** class extends the **Thread** class and implements its **run()** method.

Concerning connections and ports: The **Port** class is left out in JavaFBP. Instead of being associated with, **InputPort** is an interface which is implemented by the **Connection** class. The association between **Connection** and **OutputPort** remains.

²To prevent confusion, it should be noted that these port types are called input and output ports, respectively, from the component's point of view. From the connection's point of view inputs are outputs and vice versa.

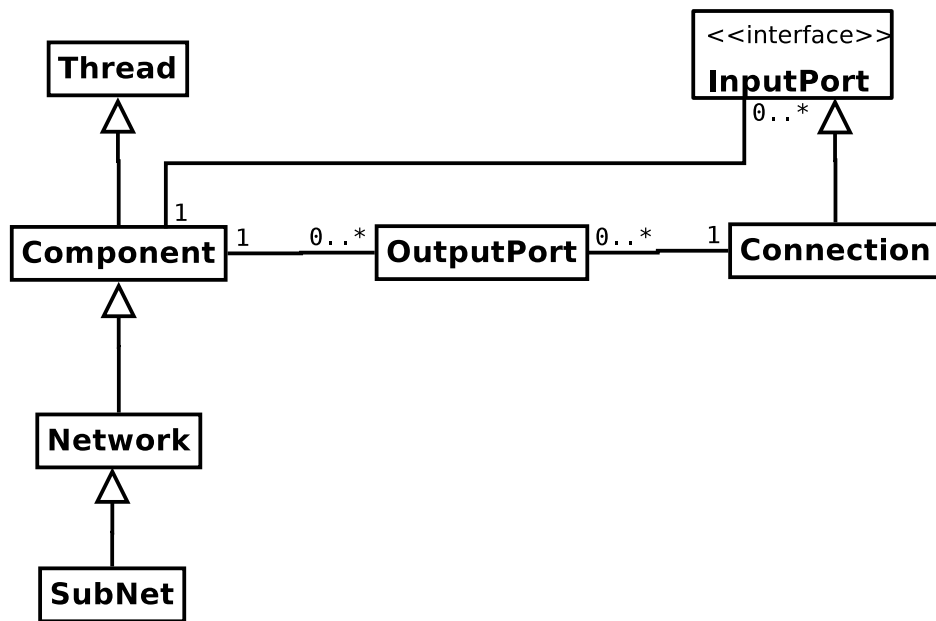


Figure 5.2: class diagram of the JavaFBP framework

5.2.2 Dynamics – Which Class Does What?

The `Network` class is responsible for

- creating the processes and connections,
- starting the processes which do not have to wait for input (see section 2.3.3),
- waiting for all processes to terminate and
- doing the deadlock detection.

The `Component` class implements the `run()` method of its superclass `Thread`. As stated in section 5.1.2, the `execute()` method contains the component's logic. When the process gets started, this method is invoked. According to the scheduling rules (section 2.3.3), the process terminates when all IPs are consumed and all input ports are closed. Otherwise, the process waits for incoming IPs and reinvokes the `execute()` method if IPs appear or terminates when the input ports get closed.

In section 2.3.2 was mentioned that connections contain a FIFO buffer accommodating IPs. This buffer is implemented as a fixed-size array of IPs in the `Connection` class (see figure 5.3). There is a pointer which is used for receiving from the connection. It points to the field of the array from which the next IP will come. On each receive, this pointer will be advanced to the next field. After pointing to the last field, it will be reset to the first field.

Similarly, there is a pointer which is used for sending to the connection. It points to the field where the next IP will go. Apart from that, it behaves like the receive pointer.

For determining whether the buffer is full or empty, an IP counter is used which always contains the number of IPs currently in the buffer.

To access the buffer, the `Connection` class offers two methods. The `send()` method is used to put IPs in the buffer and the `receive()` method is used to take IPs from the buffer. Because these methods most of the time get invoked by different threads, unless the output port of a process is connected to the input port of the same process, they are synchronized on the `Connection` object to prevent race conditions.

When the `receive()` method is invoked on an empty channel, the `wait()` method inherited from the `java.lang.Object` class is invoked to release the lock and give sending

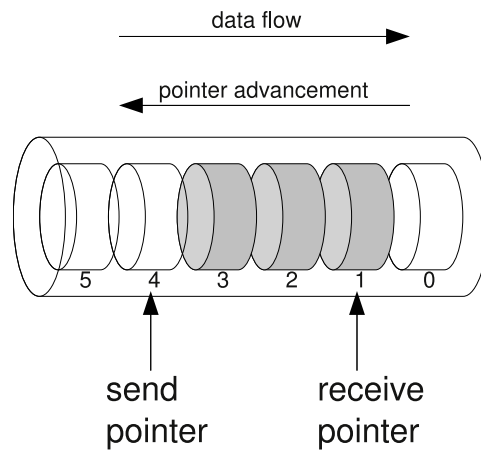


Figure 5.3: the FIFO buffer

processes the chance to put IPs in the buffer. After this has happened, the `receive()` method can take the next IP and return it. Before returning, the `notifyAll()` method inherited from the aforementioned `Object` class is called to wake up potentially waiting senders.

The `send()` method tests whether the channel is full before putting an IP in the buffer. If this is the case, it will, like the `receive()` method, invoke the `wait()` method to enable the receiver to take an IP from the buffer. As soon as there is a free slot in the buffer, the new IP will be put into this slot. Now it may be that the receiver has not been activated yet. In this case, the `start()` method of the receiving process gets invoked. Otherwise, it is notified via the `notifyAll()` method to wake it up if it is suspended on receive.

Deadlock detection is done as described in section 2.4.4: Every 1.5 (or so) seconds the states of the processes are examined. If there is no process in state “active” and some in one of the suspend states (“suspended on receive” or “suspended on send”), a deadlock is detected and the FBP program is terminated.

5.2.3 Flaws, Inconveniences and Fixes

Handling of Deadlocks

In versions of the JavaFBP library prior to 2.4 the program was terminated, in case of a deadlock, using the `System.exit()` call. Essentially, this meant telling the Java Virtual Machine (JVM) to terminate. This is no problem for doing some experiments with JavaFBP, but for using it in an application like PARASUITE, this was not acceptable. Because PARASUITE uses JavaFBP inside of a JBoss application server, calling `System.exit()` meant shutting down the whole application server. As this was not acceptable, another solution had to be created.

The idea of the solution is to tell all processes to terminate in case of a deadlock and then return from the `go()` method with an error. To determine what has to be done, one has to consider the states the processes can be in when a deadlock has occurred:

- not yet initiated
- suspended on send
- suspended on receive
- inactive
- terminated

Easy to handle are those processes which are in the “not yet initiated” state. Their `run()` method has not been invoked yet. Therefore, their state can be set to `terminate`. A check for the state at the beginning of the `run()` method ensures that they do not get activated.

With the processes which are already in the “terminated” state, nothing has to be done.

Interesting are those processes which are in one of the other three states. These processes are blocked either in a call to the `wait()` method (“suspended on send”, “suspended on receive”) or in a call to the `await()` method of a `java.util.concurrent.locks.Condition` object (“inactive”). To terminate these processes, they have to be waked up and let the `run()` method terminate. This can be done by calling the `interrupt()` method provided by the `Thread` class on them. The aforementioned blocking methods will throw an `InterruptedException`. To prevent the stack trace from being printed to the standard error output, this exception should be caught in the `run()` method. It may be, however, that this exception is thrown because an error occurred. In this case the exception should not be silently caught. To make the distinction, the process’s state should be set to `terminated` before interrupting. So the exception handler gets a hint whether the exception occurred as a result of forced termination or as a result of an error and can act accordingly.

To sum it up, when a deadlock gets detected, all processes get their state set to “terminated” and their `interrupt()` method gets called. The latter does not influence the processes which are in the “not yet initialized” or “terminated” state, because they do not run at this time.

Handling of Errors in Processes

In section 2.4.4 was stated that error handling is difficult to do in FBP. This is reflected by the fact that in JavaFBP versions prior to 2.4 error handling was absent. Instead, error messages were printed to the console, i.e., only rudimentary error reporting was present, and components terminated in uncontrolled ways leading to deadlocks and other unexpected behavior. Because this reduced the usefulness of JavaFBP for applications, an error handling mechanism had to be created. Ideally, the `go()` method should throw an exception if an error occurred during the run of an FBP network.

There are at least two possible error handling strategies. The first one could be called the “all-or-nothing” strategy. According to this strategy, either all processes of a network perform their operations without errors or they will be forced to terminate if an error occurs in any process. A second strategy, called “graceful degradation”, would be to shut down only those parts of a network which cannot function properly if an error occurs in a particular process and let the rest go on processing.

The first strategy is of particular interest during developing and testing an FBP network. In these use cases, it is often required that the system terminates as soon as possible, so that the developer can correct errors very soon. The second strategy might be more interesting when a network is used in production. Sometimes it would certainly be nice when at least partial results became available, especially when the calculations performed by the network took a long time.

The author designed and implemented the all-or-nothing strategy for the JavaFBP framework as follows:

The goal is to let all processes of a network shut down when an error occurs within a process. Thus, it has to notify the others of its error. Normal connections between processes are unsuitable for propagating error signals because they have to be explicitly listened on. If no error occurs, such a listen would cause each process to block on the “error” channel. Furthermore, the component designer would have to explicitly embed these receives into the component’s logic. Because of these issues, error handling should be done by the JavaFBP framework.

This leads to the question: How does the component signal an error to the framework? In other FBP implementations the equivalent to the `execute()` method would return a special error code to the framework [Morrison, 1994, p. 122]. According to the Java conventions for signaling errors, the `execute()` method should throw an exception in case of an error. Because the `execute()` method can be implemented using arbitrary code, probably involving third-party libraries, the type of exceptions which may occur can not be predicted. Thus, a suitable base class has to be used in the `throws` declaration. The choice fell on the `java.lang.Exception` class. Most “normal” exceptions inherit from this class. A premature choice would have been to use the `java.lang.Throwable` base class. But this would have included, in addition to the exceptions, classes inheriting from `java.lang.Error`. Errors, in the sense of inheriting from this class, are mostly abnormal conditions like virtual machine errors which cannot reasonably be handled by application code. So it made no sense to the author to try to handle these errors by the framework.

If an exception occurs in the `execute()` method, the process terminates. But that alone will not necessarily stop other processes. They have to be signaled that they should also terminate. How can this be done? – A network can be considered as a tree. The root network is the root node, instances of subnets are non-leaf nodes, and instances of normal components are leaf nodes. Every network, be it the root network or a subnet, only knows its direct descendants, i.e. those processes which appeared in its definition. On the other hand, each child node knows its direct mother. The idea is to pass an error signal through to the root node and let the root node recursively ask its child nodes, down to the leaf nodes, to terminate.

This idea is realized by a simple chain of method calls. The process in which an error occurs calls the `signalError()` method of its mother network passing the exception as an argument to this method. If the mother network is a subnet, it behaves like the component, in which the error occurred, i.e. it calls its mother. This happens until the root network is reached. The root network accommodates the exception in an instance variable (for throwing it in the `go()` method and calls its childrens’ `terminate()` method which is responsible for terminating the called node. The `terminate()` method of a subnet calls the respective methods of its direct child nodes.

An illustration of this chain of method calls is given in figure 5.4. An error occurs in “process4”. Because it is a child process of “subnet1”, it calls the `signalError()` method of this subnet instance which in turn calls the `signalError()` method of its “mother”. This happens to be the root network. The first phase which propagates the error signal is finished. Now the second phase starts which is responsible for propagating the termination signal to all processes starting from the root network.

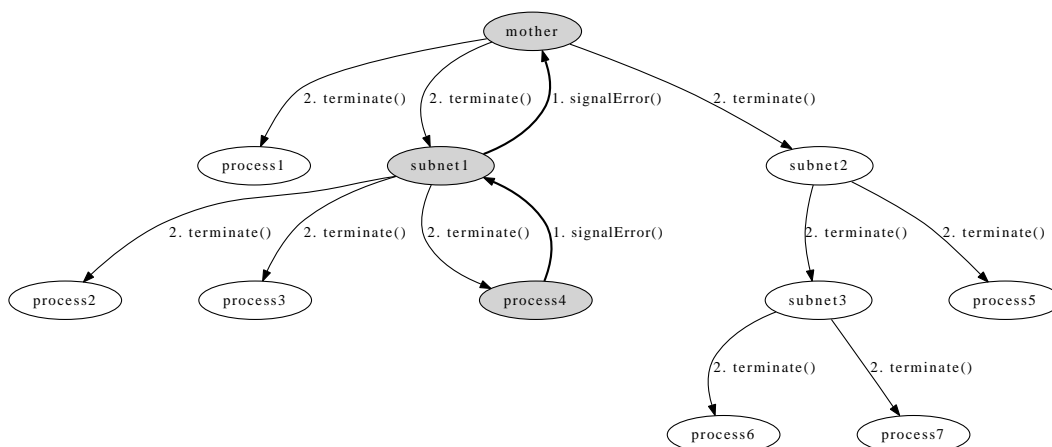


Figure 5.4: illustration of the method-call chain in case of an error

Java provides a method (`Thread.stop()`) to immediately stop threads. But this method is deprecated due to various reasons (see the documentation in Sun Microsystems a). As with handling deadlocks, the `interrupt()` method is used instead. The `OutputPort.send()` and `InputPort.receive()` methods will react eventually throwing an `InterruptedException` which should terminate the process. Unfortunately, this technique does not guarantee a certain response time. Especially, when the process does some lengthy processing without interacting with the ports (sending or receiving), it may take a very long time until the process terminates. Component developers should therefore check the `interrupt` flag with the `interrupted()` or `isInterrupted()` of the `Thread` class during lengthy calculations and let the process terminate.

6 Scalability of JavaFBP

In PARASUITE it is not unusual to have FBP networks with approximately 500 processes in use. As these networks are intended to be designed and extended by end users, it is not clear how much processes will be used in the future. In this section it shall be examined whether the current Java implementation of FBP is suitable for this type of application in regard to several aspects. In short, there are two main questions:

- How many processes can be used in a FBP network? – If not many more processes than 500 can be used within JavaFBP, the users may soon hit the limits which is not desirable.
- How does JavaFBP scale with the number of processes? – If JavaFBP allows using much more processes but the runtime increases exponentially with the number of processes, this virtually has the same effect as limiting the processes to a certain number.

If the results show that there are limitations, possible causes shall be examined and some solutions helping to overcome or at least mitigate these limitations will be proposed.

6.1 The Testing Environment

The server part of PARASUITE is meant to run on server-class machines with more than one CPU and several gigabytes of memory. For the conduction of the tests a virtualized server with the following specifications is used:

- CPU: Intel Xeon E5345 2.33 GHz with four cores
- Memory: 4 GB

The underlying hardware platform is composed of a cluster of four IBM System x3650 servers with

- two Intel Xeon quad-core CPUs,
- 18 GB main memory

per single server.

The virtualization is done by VMware Infrastructure 3.5 Enterprise. Normally, virtualization software provides resources like CPU processing power and memory on demand and according to the availability of the resources. For the tests, however, a predictable and constant set of resources, independent from the overall load of the underlying hardware, is needed. To achieve this, the VMware virtualization software was configured to provide a fixed CPU frequency and a fixed amount of memory as given above to the virtualized server.

The (guest) operating system under which the tests were conducted was Ubuntu Linux 8.10 32 bits providing a Linux kernel in version 2.6.27-11-generic. Sun's Java runtime environment (JRE) in version 1.6.0 Update 10 was used to run JavaFBP. The version of the latter was revision 155 from the JavaFBP subversion repository plus some bug fixes and minor modifications.

According to Sun Microsystems [b], the provided machine is considered as a “server-class” machine by the JRE because there is more than one CPU (core) and more than two GB of memory available. This affects some default settings of the JRE: The initial heap size is set to 1/64 of the physically available memory as reported by the operating system, and a maximum heap size is 1/4 of the physically available memory is reserved. Additionally, the “Server” virtual machine (VM) is used which brings some optimizations for large applications, and the parallel garbage collector is used which is said to be optimized for maximum throughput.

Just to be sure, the `-server` and `-XX:+UseParallelGC` command line options were used to select the Server VM and the parallel garbage collector in all invocations of the JRE.

6.2 Network Topologies

6.2.1 The Serial Topology

For the tests three distinct network topologies were used. The first one is called the “serial” topology because all involved components are connected serially. A serial network consists of a data source, a data sink and a configurable number of calculating processes in between as shown in figure 6.1.

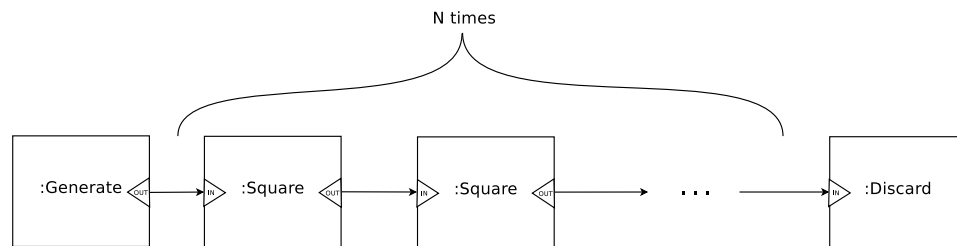


Figure 6.1: serial network topology

The data source component is called **Generate** and implemented as follows (complete listing in appendix A.2.7):

```

1  package components ;
2
3  @InPort ("PACKETS")
4  @OutPort ("OUT")
5  public class Generate extends Component {
6
7      private InputPort packetsPort ;
8      private OutputPort outputPort ;
9
10     @Override
11     protected void execute() throws Exception {
12         // read the number of IPs to be produced
13         Packet conf = packetsPort.receive ();
14         int numberOfPackets = (Integer) conf.getContent ();
15         drop(conf);
16         packetsPort.close ();
17
18         // create and send the IPs
19         for (int i = 0; i < numberOfPackets; i++) {
20             outputPort.send(create(i));
21         }

```

```

22     }
23
24     @Override
25     protected void openPorts() {
26         packetsPort = openInput("PACKETS");
27         outputPort = openOutput("OUT");
28     }
29 }

```

This component does the following: It reads the number of IPs (n) to be produced from the `PACKETS` port (lines 13 and 14) and outputs n IPs containing the values 0, 1, ..., $n-1$ (lines 19-21).

The component used for calculating processes is called `Square` and looks like this (complete listing in appendix A.2.9):

```

1 package components;
2
3 @InPort("IN")
4 @OutPort("OUT")
5 public class Square extends Component {
6     private InputPort inputPort;
7     private OutputPort outputPort;
8
9     @Override
10    protected void execute() throws Exception {
11        Packet p;
12        while ((p = inputPort.receive()) != null) {
13            int i = (Integer) p.getContent();
14            drop(p);
15            // calculate the square and send it to OUT
16            outputPort.send(create(i*i));
17        }
18    }
19
20    @Override
21    protected void openPorts() {
22        inputPort = openInput("IN");
23        outputPort = openOutput("OUT");
24    }
25 }

```

This component reads IPs from the `IN` port (lines 12 and 12) and sends the square of each IP's value to the `OUT` port (line 16).

The data sink component is the `Discard` component as shipped with the JavaFBP framework. This component simply drops each incoming IP.

6.2.2 The Parallel Topology

The second network topology is the so-called “parallel” topology (see figure 6.2). Networks of this topology consist of a configurable number of small, independent networks with a `Generate` process connected to a `Square` process which, finally, is connected to a `Discard` process.

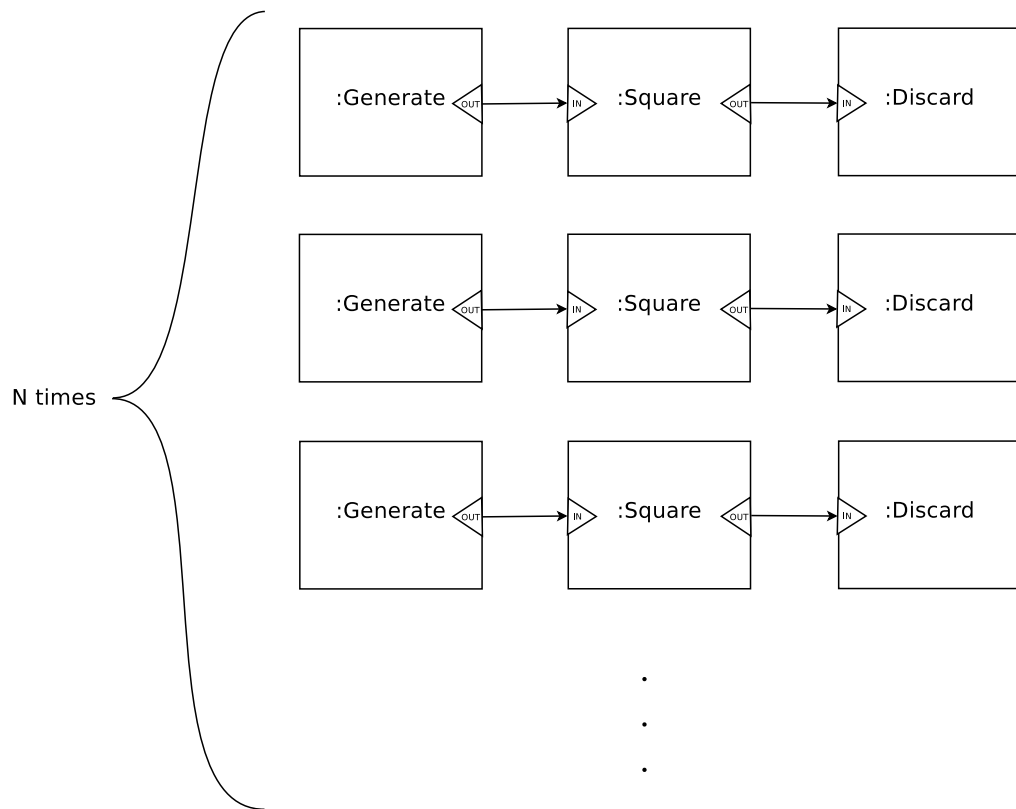


Figure 6.2: parallel network topology

6.2.3 The Triangle Topology

The third network topology is called “triangle”. This has two reasons: First, the topology looks like a triangle, and second, networks of this topology calculate Sierpinski triangles (figure 6.3) of configurable depth.

Sierpinski triangles are fractals which can be generated according to the following algorithm:

1. Draw a filled triangle.
2. Connect the midpoints of the sides.
3. Remove the resulting triangle.
4. For each of the three resulting triangles, start again at step 2.

Networks with the triangle topology (figure 6.4 shows an example of a two-layer triangle network) consist of a data source called `TriangleGenerator` which generates a configurable amount of IPs containing triangle descriptions. These descriptions consist of the coordinates of the three vertexes (A, B and C) and a triangle id. The output of this data source is fed into another process which is an instance of `SierpinskiSelector`. This process has three output ports: `LEFT`, `RIGHT` and `TOP`. At each of these ports a sub-triangle will be outputted as required by the above construction algorithm, the triangle at vertex A at `LEFT`, the one at B at `RIGHT`, and the one at C at `TOP`. According to the configured depth, a layer of `SierpinskiSelectors` is connected to these ports. The final layer is connected to a `Discard` process.

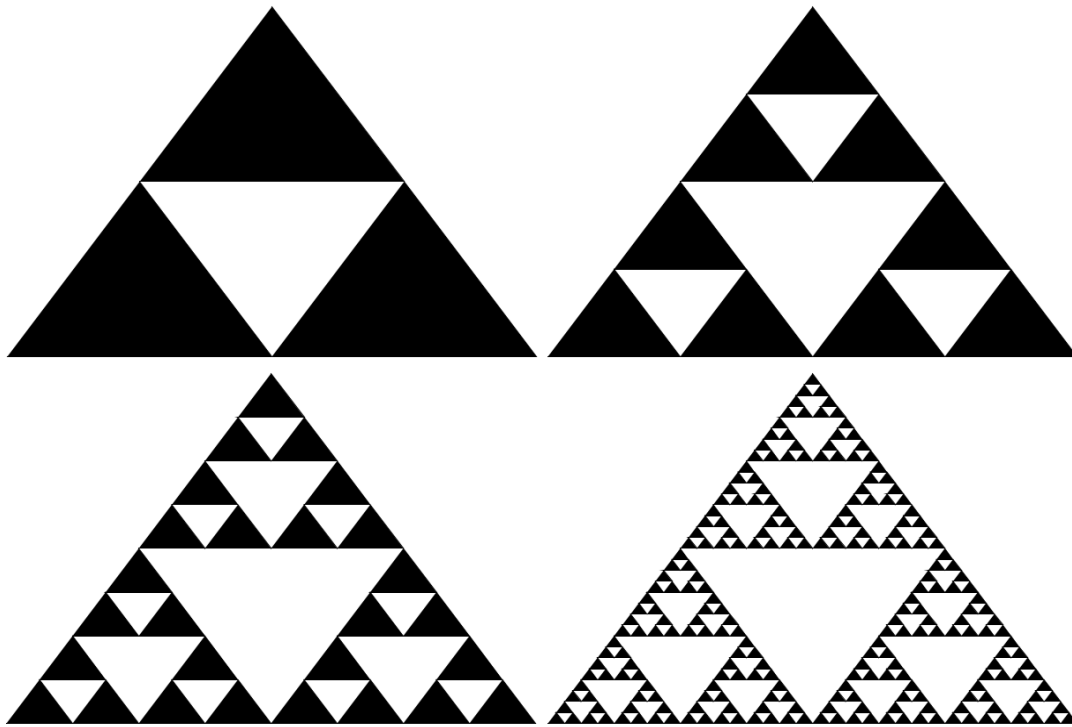


Figure 6.3: Sierpinski triangles of different depths – 1 (top left), 2 (top right), 3 (bottom left), 5 (bottom right)

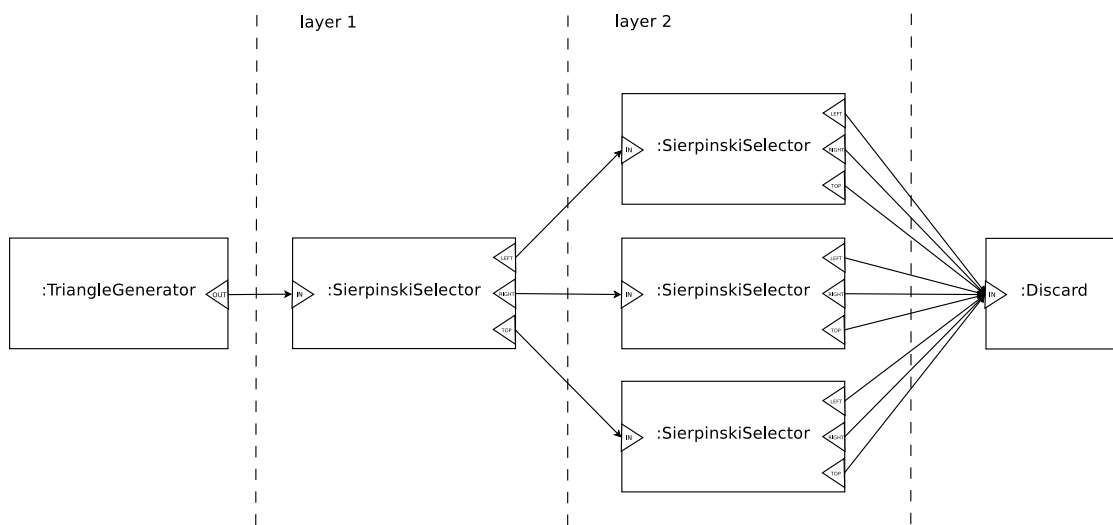


Figure 6.4: a two-layer triangle network

6.2.4 Rationale

All three network topologies have in common that data is processed in memory, routed through some calculating processes and, finally, discarded in one or more processes. In real applications (like PARASUITE) data will come from a database or a file and the results of the calculations will be written to a database or a file. The reason to leave out such data sources and sinks is to not have disc access influence the test results. Purposely, only the CPU and main memory behavior of JavaFBP should be measured.

Three different network topologies have been chosen to test how the thread scheduler behaves in different situations. The serial topology can be used to force the scheduler to have all threads busy at the same time, the parallel topology leaves room to schedule parts of the network at different times because not all processes depend on each other. Finally, the triangle topology is a mixture between the two former topologies. While the distinct layers are connected serially, the processes in each layer are not that dependent on each other. This type of network topology, additionally, may be closest to networks used for real task, because often there are few data sources and few data sinks but many calculating processes in between, splitting and routing data in different directions. A special property of this topology is that the amount of data grows in each layer. In each instance of `SierpinskiSelector` three IPs are generated for each incoming IP. In the other topologies, however, the amount of data remains constant.

6.2.5 Test Program

To conduct the following tests, a test program (see A.2 for the source code) was created by the author which can create and run networks of the mentioned three network topologies in different sizes. It is packaged in a .jar file which can be run directly by the JRE. It expects at least three command line arguments:

1. the topology – either `serial`, `parallel` or `triangle`,
2. the number of IPs,
3. the network size

As fourth argument the size of the connection buffers can be specified. It defaults to ten. With the fifth argument, some flags can be specified: If it contains the phrase `nowarmup`, no warm-up phase is done (see section 6.4 for details). If it contains the phrase `single`, it only performs a single run (see the same section for the default behavior).

Depending on the specified topology, the second and third arguments have a different meaning:

For the serial topology, the number of IPs specifies how many IPs should be created by the `Generate` process. The network size specifies how many `Square` processes shall be created. Because each serial network has a data source and a data sink process, the overall number of processes in the network is always the network size plus two.

For the parallel topology, the network size specifies how many process triples consisting of a `Generate`, a `Square` and a `Discard` process shall be created. This means that the number of processes is equal to the network size multiplied by three in each parallel network. The number of IPs specifies how many IPs are created by the `Generate` processes. This means that the overall number of IPs which are created in those processes is the number of IPs multiplied by the network size.

Finally, for the triangle topology, the number of IPs specifies how many IPs are generated by the single `TriangleGenerator` process. The network size specifies how many layers of `SierpinskiSelector` processes are created. A triangle network always contains a data source and a data sink process. In the first layer there is a single `SierpinskiSelector` process. In each of the following layers there are three times as much `SierpinskiSelectors` as in the previous layer. Thus, the number of those

processes follows a geometric series. If x is the network size, the number of processes $N(x)$ in a triangle network can be calculated as follows:

$$N(x) = 2 + \sum_{i=0}^{x-1} 3^i$$

This results in the following number of processes for network sizes from one to ten:

network size	number of processes
1	3
2	6
3	15
4	42
5	123
6	366
7	1095
8	3282
9	9843
10	29526

6.3 How many processes does the JRE allow?

The first tests are intended to find out how many JavaFBP processes can be created and run by the JRE. Additionally, it should be examined what the limiting factors are.

In general, space for Java objects is allocated on the heap. This is also true for component objects representing FBP processes. On the other hand, the `Component` class inherits from `Thread`. Java threads are implemented using operating system threads on Linux. On invoking the `start()` methods of the `Thread` objects, an operating system thread is created and started which executes the `run()` method. This means, creating a `Thread` object does not directly result in creating an operating system thread and reserving the associated resources. When threads are created, a certain amount of space is allocated for the stack.

With this in mind, the following hypotheses can be assumed:

Hypothesis 1 *The number of processes in networks with few active processes at the same time will be constrained by the memory which can be allocated for the heap.*

Hypothesis 2 *The number of processes in networks with many active processes at the same time will be constrained by the memory which can be allocated for the stack.*

In this context “active” means, that a process has already entered the `run()` method but not left it. I.e. stack memory is allocated for this process.

To test the first hypothesis the author created and run serial networks with a connection capacity of one IP – to prevent the connections to take up too much space – and let a single IP travel through the networks. He increased the number of processes until the JRE terminated with the following error:

```
Exception in thread ‘main’ java.lang.OutOfMemoryError: GC overhead
limit exceeded
```

This happened using a network with 700002¹ processes. A smaller one with 600002 processes would run. According to Sun this error may occur when the heap is too small,

¹These strange numbers occur because there are always two more processes in a serial network than specified on the command line – see section 6.2.5.

which hints that hypothesis 1 may be true. If it is true, increasing the heap size would allow to use at least 700002 processes. To verify this, the author increased the maximum heap size using the `-Xmx2600m` command line switch which sets the maximum heap size to 2600 megabytes (more was not allowed by the JRE: “Could not reserve enough space for object heap” was the error message). With this setting the usage of 1600002 processes was possible. This proves that hypothesis 1 is true.

To test the second hypothesis a network using the following properties was used:

- topology: serial
- number of IPs: more than the number of connections
- connection capacity: 1

The rationale behind using more IPs than there are connections is to have all processes active at the same time. Thus, the maximum number of memory allocated for the stack is used.

With the default memory settings, only 5002 processes could be active at the same time. With 6002 processes the following error occurred:

```
Exception in thread ‘‘inner5680’’ java.lang.OutOfMemoryError: unable to
create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:597)
```

This message reveals the following: Indeed, no operating system thread is created before calling the `Thread.start()` method. But the message “unable to create new native thread” does not say whether a memory or operating system limit was hit (although the type of exception, `OutOfMemoryError`, might indicate hitting a memory limit).

The JRE allows the stack size allocated per thread to be set using the `-Xss` switch. If we hit a memory limit, reducing the stack size would result in being able to run more threads at the same time because more thread stacks would fit in the same amount of memory. If we hit an operating system limit, the results should be the same.

The tests revealed the following: Setting the stack size to the lowest possible value, 64 kilobytes (`-Xss64k`), enabled JavaFBP to have 23002 processes active at the same time. This proves that hypothesis 2 is true.

Another interesting question is: How does the reserved heap size influence the number of active threads? – Tests revealed that increasing the maximum heap size to 2600 megabytes (`-Xmx2600m`) decreased the possible number of active threads to 502 (with default stack size). Obviously, memory reserved for the heap can not be used by the JRE to allocate stack memory.

Using this findings, one can predict, how networks of the parallel and triangle topologies will behave.

In networks with a parallel topology, one third of the processes, namely the `Generate` processes, will be started one after another by the main thread, in which the `go()` method was called. Because each process triple consisting of a `Generate`, a `Square` and a `Discard` process will only handle a single IP and then terminate, processes will probably run quickly and soon will be disposed of. Not many processes will run in parallel. Therefore a similar behavior may be expected as with a serial network.

Indeed, the test results reflect this:

1. default settings, connection size = 1, 1 IP per process triple \Rightarrow 600000 processes possible
2. 2600 MB heap, connection size = 1, 1 IP per process triple \Rightarrow 1800000 processes possible

3. default settings, connection size = 10000, 10000 IPs per process triple \Rightarrow 6000 processes possible
4. 64 kB stack size, connection size = 10000, 10000 IPs per process triple \Rightarrow 21000 processes possible

As stated above, the main thread is responsible for starting most of the processes. Each **Generate** process will not start the connected **Square** process before doing the first **send** (see section 5.2.2), which in turn will start the connected **Discard** process. It looks like the main thread is able to start some process triples but soon will be suspended by the scheduler to let some of the newly created processes get the processor. Because processing a single IP is done very fast by a process triple, some processes will have been terminated by the time the main thread has the chance to start some more processes. Obviously, this does prevent the number of active processes to reach the limit where available stack memory is an issue.

As shown by the third test results, this effect does not occur when a process triple has much to do (i.e. processing 10000 IPs). Then the main thread obviously has the chance to start more processes before the first process triple has terminated.

Applying this knowledge to triangle networks: The main thread only starts the single **TriangleGenerator** process. All other processes depend on incoming IPs. Each **SierpinskiSelector** process which does not belong to the last layer will start three other ones. Even if only one IP has to be processed by a single process, at a certain stage so many processes will be started that a significant amount of them have to wait for the processor. Therefore, it may be assumed that this topology will hit the memory limit for the stack. As could be seen with the serial topology, this limit lies between 5000 and 6000 processes. As stated in section 6.2.5, a network with eight layers has 3282 processes and a network with nine layers has 9843 processes. The former does not hit the critical number of processes, so it can be assumed that it will run, but the latter is far beyond the limit. Thus, it will likely hit the memory limit.

The results confirm this: Using the default settings, a connection capacity of one IP and a single start IP, the eight-layer network did run while the nine-layer one crashed with a “unable to create new native thread” error.

To conclude, on the test machine, the number of processes which can be active at the same time lies between 5000 and 6000, but a network may consist of several orders of magnitude more processes if the number of active ones at a time stays below 6000. Furthermore, the network topology can influence the number of active processes at a time. The allowed number of active processes can be increased by decreasing the amount of memory allocated for a single thread stack. The allowed number of process objects can be boosted by increasing the maximum heap size of the JRE.

6.4 How does JavaFBP scale?

In this section, the author wants to look into the question, how good JavaFBP scales in regard to runtime performance with an increasing number of processes. In the tests, the three already discussed network topologies were evaluated. The tests were performed the following way: For each topology, networks of different sizes (from small networks of 10 processes up to networks of several thousand processes) were run and their run time measured and compared.

JavaFBP would scale ideally if the doubling of the processes resulted in doubling the run time, i.e. if the run time scaled linearly with the number of processes.

6.4.1 Statistically Rigorous Performance Evaluation

Georges et al. [2007] point out, that it is not easy to benchmark Java programs, because

the same code very likely does not take the same time on each run. According to them this is caused by several factors:

- Just-In-Time (JIT) compilation,
- optimization mechanisms in the Java virtual machine (JVM),
- thread scheduling,
- garbage collection,
- ...

Especially, a just started JVM behaves differently than one which has run for several minutes or hours, because at first every byte code runs in interpreted mode and not until some time passes the code of the most used methods will be compiled into native machine code by the Just-In-Time compiler. In this section, the so-called **steady-state** performance shall be measured as proposed in the paper by Georges et al. This can be done as follows:

First, the JVM is warmed up by running the same code again and again until the steady state of the JVM is reached. This is determined by running an FBP network of the same topology like the one which should be benchmarked and measuring its run time. As soon as the coefficient of variation (CoV) of the last five run times falls below 0.02, the warm-up phase is considered as finished. According to *ibid.*, p. 10, the “CoV is defined by the standard deviation s divided by the mean \bar{x} ”.

After the warm-up phase, the network to be actually tested is run five times and the arithmetic mean of those run times is calculated and printed to **System.out**.

At least ten such JVM invocations are performed, and, starting with the eleventh, the average of the reported values is calculated along with a confidence interval using a confidence level of 0.95. As soon as the confidence interval drops below 0.02 times the average, or 50 invocations are performed, the average is reported.

This reported value is considered as the run time of the network.

6.4.2 Tools

To test the three network topologies, the test program mentioned in section 6.2.5 is used. If the **nowarmup** flag is omitted, the JVM is warmed up as described above. If the **single** flag is omitted, five runs are performed instead of one, and the average run time is reported. Furthermore, only the pure run time of the networks is considered, not the time it takes to construct the networks (calling **Network.define()**).

Georges et al. provide a Python script called “JavaStats” which drives the JVM invocations and calculates the confidence interval. The script can be obtained from <http://www.elis.ugent.be/JavaStats> (Retrieved: July 10, 2009). To use JavaStats, it has to be configured using two files. The first one is a simple configuration file which specifies the test to be run and some other properties like the confidence level, the maximum size of the confidence interval, etc. The configuration file used in the tests can be found in appendix A.1.1. The second file is a Python source file containing a class which is responsible for parsing the output of the Java program under test (see appendix A.1.2).

To plot the results, “gnuplot” was used (<http://www.gnuplot.info>).

6.4.3 Serial Networks

The first tests evaluate how JavaFBP scales when networks with a serial topology are run. To begin with, a serial network is tested which processes a single IP and uses a connection size of ten. In the following tests always a connection size of ten is used

because this is the default for JavaFBP and, therefore, likely reflects the setting used in real applications.

The test is run for several different network sizes. The step size differs for different orders of magnitude. From 12 to 102 processes, a step size of ten is used, from 102 to 1002, the step size is 1000 and so on. This is done to get some values from each order of magnitude without needing to run too many tests.

In the following tables, for each number of processes the average run time is given. For those cases where the size confidence interval is above two percent, it is reported in the third column.

Here are the result for a serial network processing a single IP:

number of processes	run time/ms	size of confidence interval/percent
12	5	21
22	10	29
32	14	6
42	19	14
52	24	11
62	29	13
72	34	14
82	36	4
92	44	11
102	47	10
202	96	5
302	143	5
402	198	9
502	243	6
602	278	3
702	333	6
802	395	6
902	435	4
1002	464	4
2002	958	3
3002	1404	
4002	1915	2.1
5002	2346	
6002	2889	
7002	3301	
8002	3779	
9002	4277	
10002	4729	

Because the raw numbers are hard to interpret, some plots shall be presented. In all of the following graphics, the run time is plotted against the number of processes.

The first figure (6.5) shows the complete results for the test series. Because of the scale of the plot, the dots of the small numbers of processes (from 12 to 102) can not be distinguished from each other. But the run time appears to increase linearly with the number of the processes. The points almost exactly lie on a straight line.

If we zoom in, so that only numbers from 12 to 1002 or 12 to 102 are visible (figures 6.6 and 6.7), the linearity gets confirmed although there are some slight bumps in the graph. But these can be attributed to the fact that the desired narrowness of the confidence intervals has not been reached for this ranges.

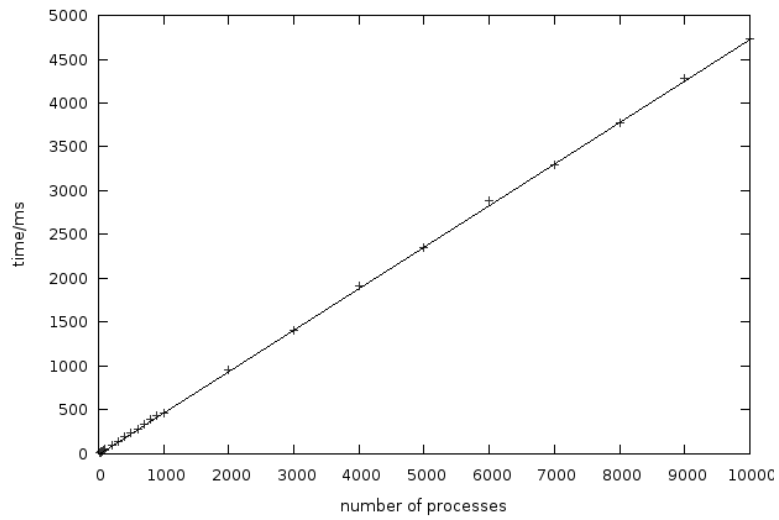


Figure 6.5: serial network, 1 IP, 12 to 10002 processes

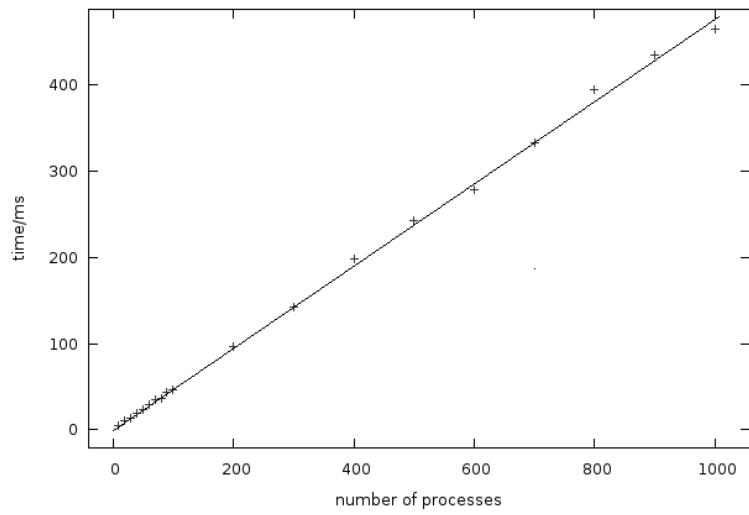


Figure 6.6: serial network, 1 IP, 12 to 1002 processes

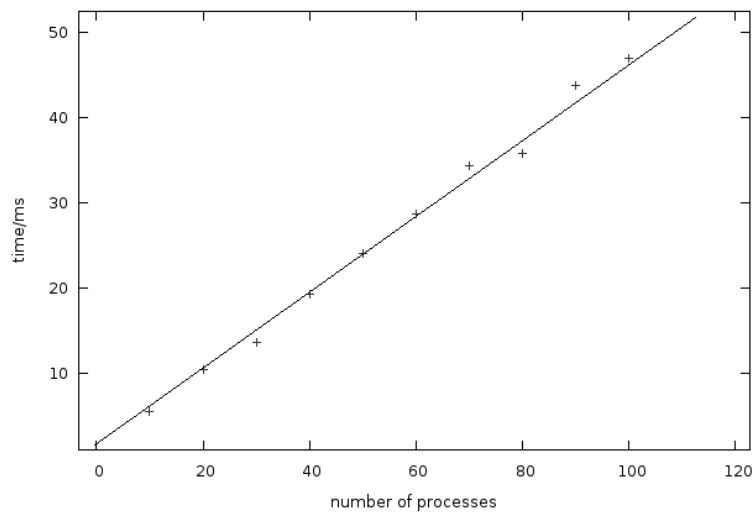


Figure 6.7: serial network, 1 IP, 12 to 102 processes

Testing the same network, but with generating 10000 IPs, yields the following results:

number of processes	run time/ms	size of confidence interval/percent
12	1233	3
22	2090	
32	2837	
42	3620	
52	4444	
62	5214	
72	5987	
82	6818	
92	7543	
102	8470	
202	16874	
302	25730	
402	35185	
502	44266	
602	53769	
702	64487	
802	74578	
902	87648	
1002	97440	
2002	213400	
3002	351172	
4002	504325	
5002	674109	

The plot of these values (figure 6.8) shows that the graph runs in a slight bow below a auxiliary line drawn through the first and the last point. This bow is still visible in the zoomed in plot in figure 6.9 with the number of processes between 12 and 1002. If it is zoomed in so that it only shows the range from 12 to 102 (figure 6.10), this effect cannot be seen any more. The points lie quite exactly on a straight line in this range. Because there is no sharp bend anywhere, it is difficult, if not impossible, to say from which number of processes on the system does not scale linearly anymore. Since the bow in the first figure does not deviate much from the straight line, one can not say that JavaFBP scales bad in the tested range.

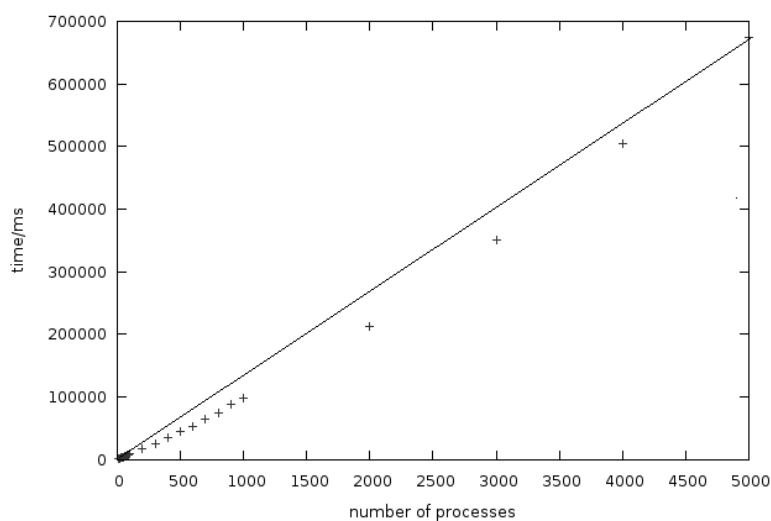


Figure 6.8: serial network, 10000 IPs, 12 to 5002 processes

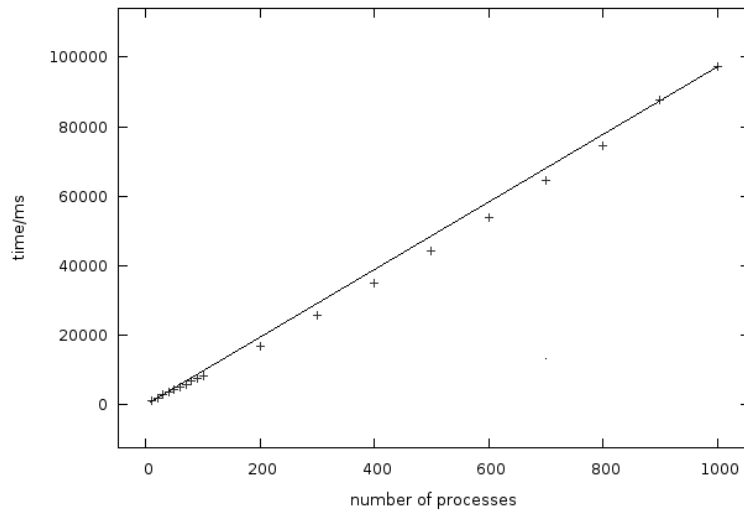


Figure 6.9: serial network, 10000 IPs 12 to 1002 processes

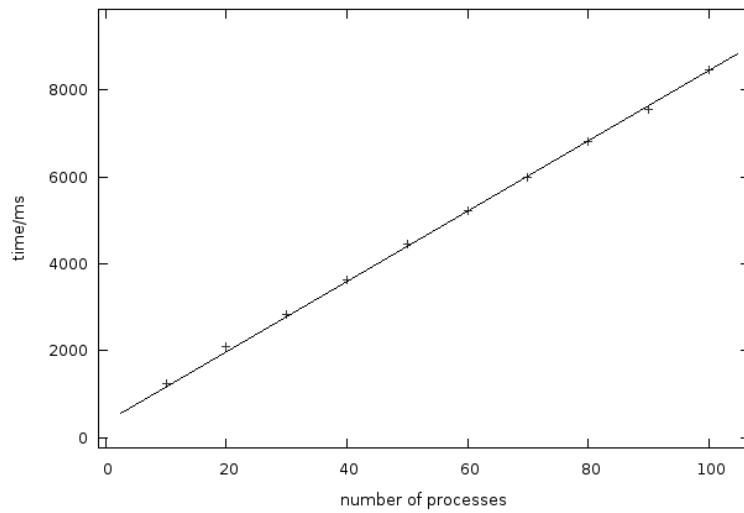


Figure 6.10: serial network, 10000 IPs, 12 to 102 processes

6.4.4 Parallel Networks

Testing of networks with a parallel topology is very much done the same way as with the serial ones. Only the step size is 30 processes rather than 10. Here are the results for parallel networks processing a single IP in each process triple:

number of processes	run time/ms	size of confidence interval/percent
30	9	11
60	20	47
90	27	6
120	36	8
150	45	10
180	55	14
210	61	5
240	73	11
270	80	8
300	87	3
600	177	4
900	264	6
1200	381	22
1500	455	7
1800	542	4
2100	628	4
2400	719	4
2700	803	3
3000	897	2
6000	1804	3
9000	2673	
12000	3612	6
15000	4495	
18000	5333	
21000	6356	5
24000	7280	4
27000	7756	
30000	8935	
60000	18301	
90000	27524	
120000	36887	
150000	45412	
180000	54595	
210000	63842	
240000	72817	
270000	82461	
300000	90783	
600000	183074	

The plots of these values (figures 6.11, 6.12, 6.13, 6.14 and 6.15 look very similar than those of the serial networks processing a single IP. Parallel networks scale linearly in the tested range when only a single IP has to be processed by a process triple.

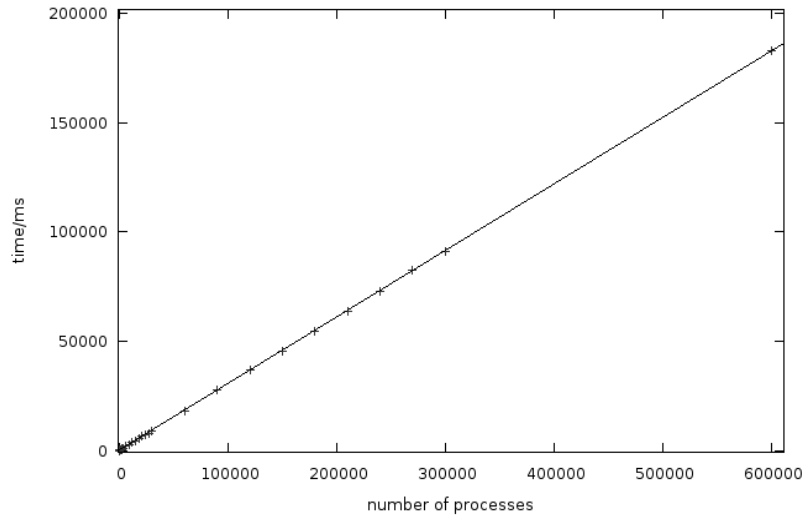


Figure 6.11: parallel network, 1 IP, 30 to 600000 processes

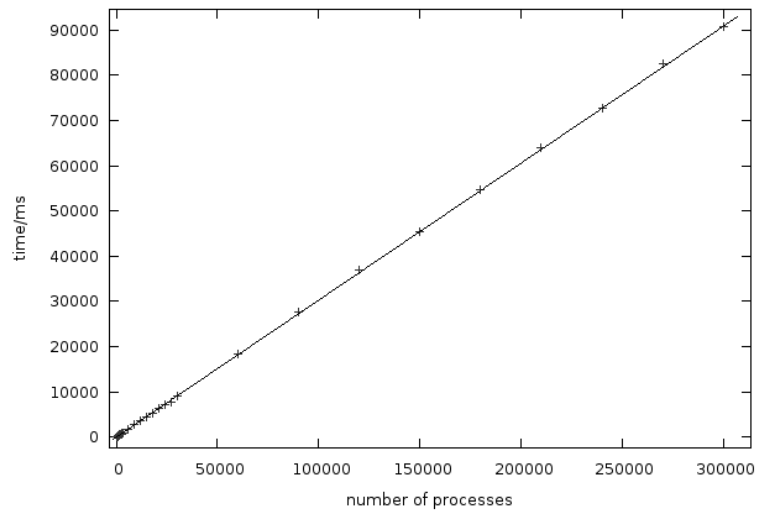


Figure 6.12: parallel network, 1 IP, 30 to 300000 processes

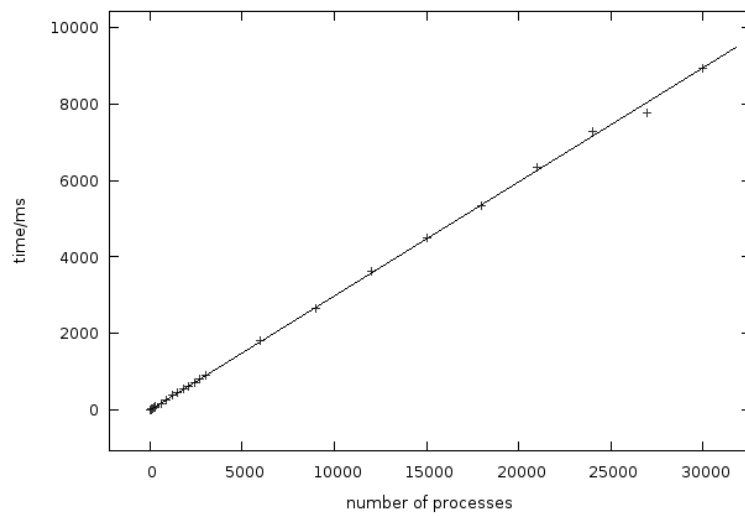


Figure 6.13: parallel network, 1 IP, 30 to 30000 processes

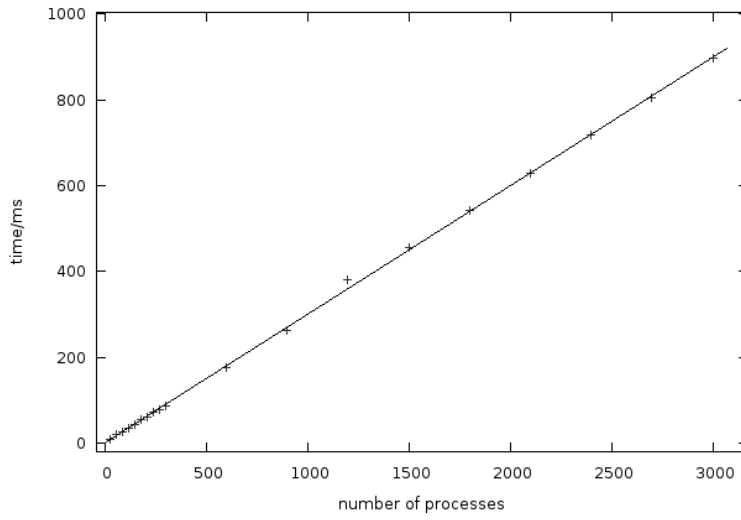


Figure 6.14: parallel network, 1 IP, 30 to 3000 processes

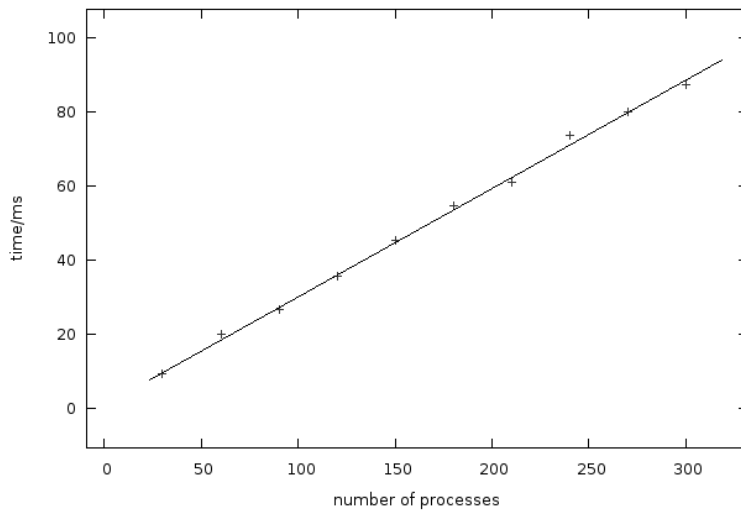


Figure 6.15: parallel network, 1 IP, 30 to 300 processes

Testing parallel networks where each process triple processes 10000 IPs yields the following results:

number of processes	run time/ms	size of confidence interval/percent
30	3754	6
60	5938	
90	8488	
120	11025	
150	13703	
180	16311	
210	18911	
240	21802	
270	24857	4
300	27523	4
600	56767	
900	86836	
1200	116018	
1500	148098	
1800	185735	
2100	220814	
2400	263920	
2700	298352	
3000	343764	

Looking at the plots (figures 6.16 and 6.17) reveals a very similar scaling behavior like in the respective serial test case. Also in this case JavaFBP scales a little worse than linearly.

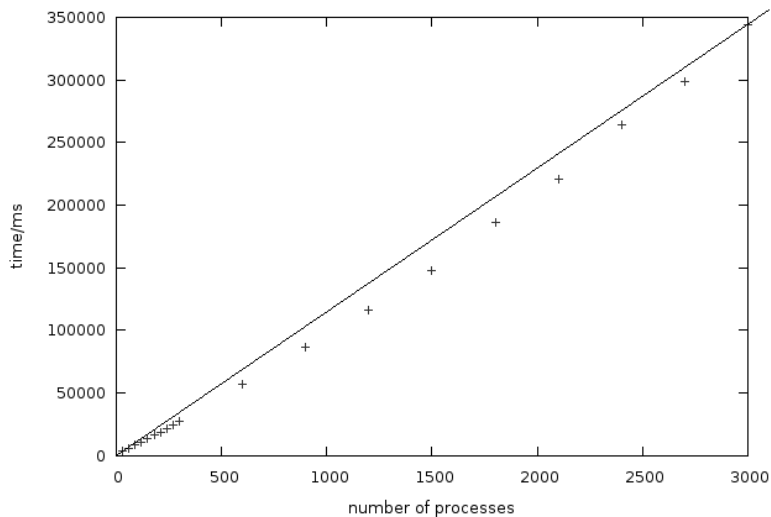


Figure 6.16: parallel network, 10000 IPs, 30 to 3000 processes

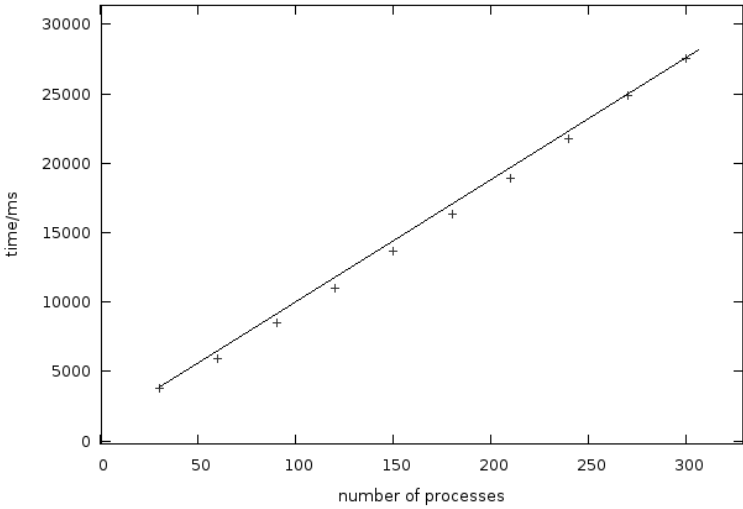


Figure 6.17: parallel network, 10000 IPs, 30 to 300 processes

6.4.5 Triangle Networks

Finally, networks with a triangle topology are tested. In this case, the step size is determined by the number of layers. Tested are networks from 1 to 8 layers. Here are the results for a network processing a single generated IP:

number of processes	run time/ms	size of confidence interval/percent
3	3	69
6	4	22
15	7	27
42	17	14
123	47	13
366	174	7
1095	930	5
3282	6354	6

The first three plots of the data (6.18, 6.19 and 6.20) show that JavaFBP does not scale in a linear fashion with increasing numbers of processes. Only between 3 and 123 processes linear scaling can be recognized (6.21).

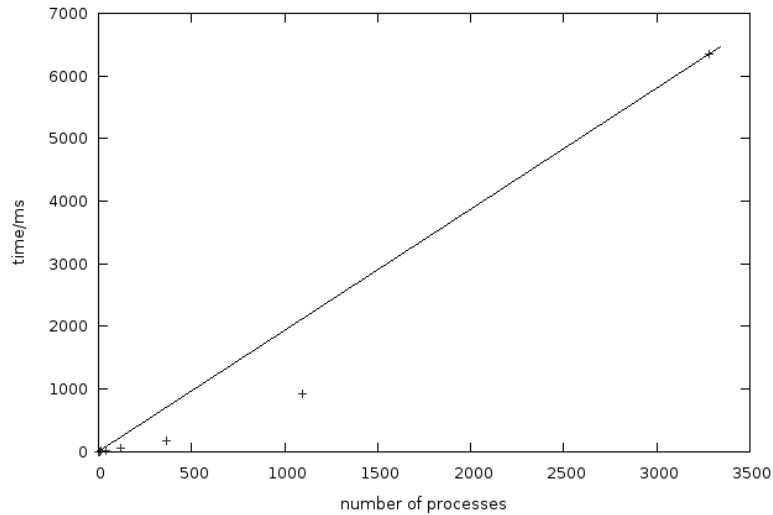


Figure 6.18: triangle network, 1 IP, 3 to 3282 processes

When 10000 Sierpinski triangles shall be calculated, the results look like the following:

number of processes	run time/ms	size of confidence interval/percent
3	836	8
6	2252	
15	8096	
42	33858	
123	131413	
366	627621	

As can be seen in figure 6.23, with 10000 IPs the linear scaling is gone even between 3 and 123 processes.

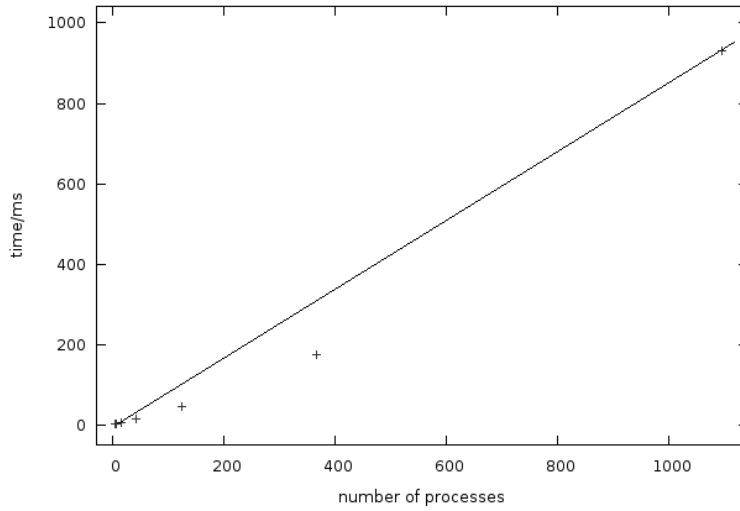


Figure 6.19: triangle network, 1 IP, 3 to 1095 processes

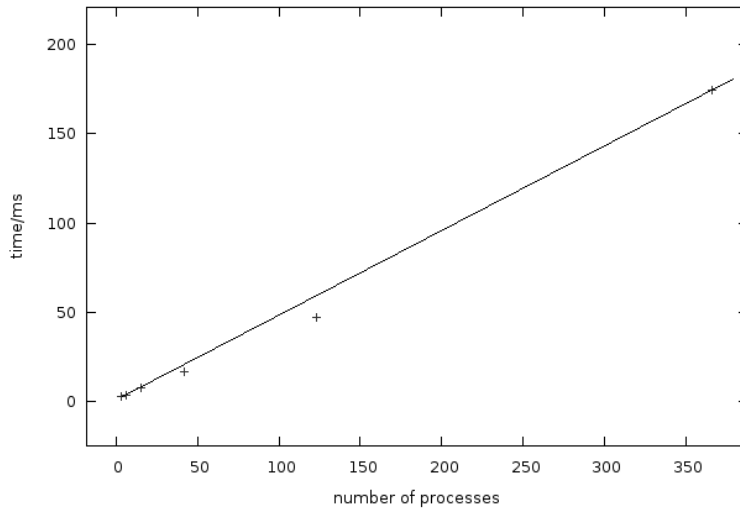


Figure 6.20: triangle network, 1 IP, 3 to 366 processes

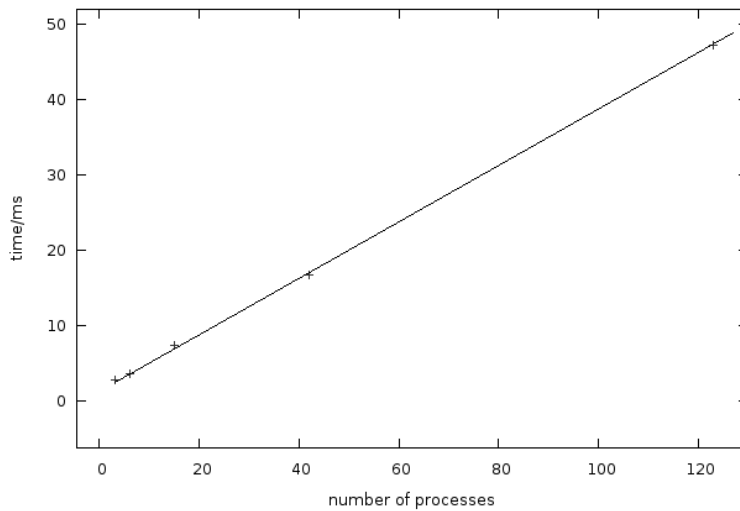


Figure 6.21: triangle network, 1 IP, 3 to 123 processes

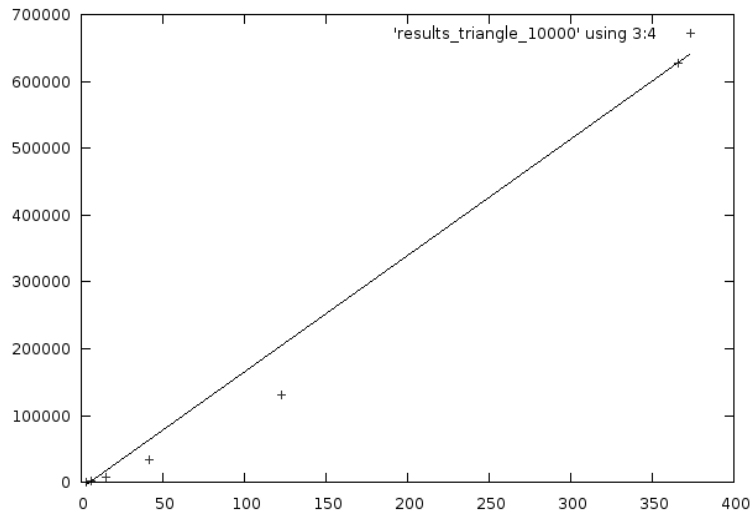


Figure 6.22: triangle network, 10000 IPs, 3 to 366 processes

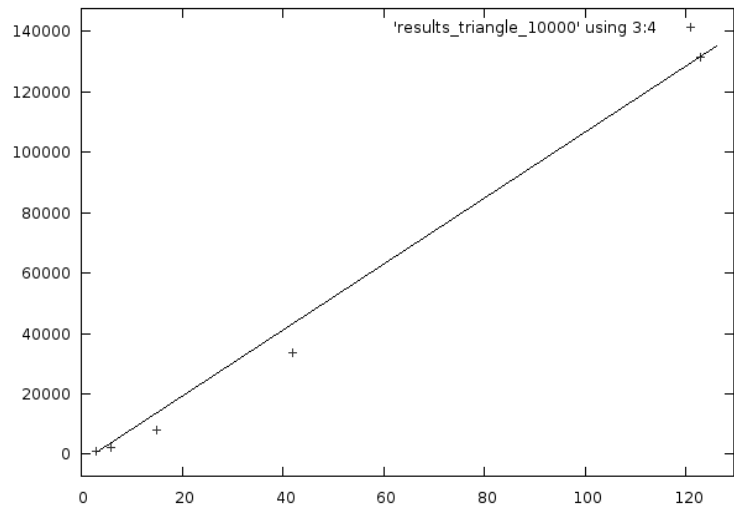


Figure 6.23: triangle network, 10000 IPs, 3 to 123 processes

6.4.6 Conclusions

To sum it up, when the amount of data to be processed is low and not many processes are active at the same time, JavaFBP scales very well, namely in a linear fashion, with increasing numbers of processes. But as soon as more processes are active at the same time (caused by the amount of data in the serial and parallel cases or by the network topology in the triangle case), JavaFBP does not scale in a linear way anymore. Instead, when increasing the number of processes, the execution takes disproportionately more time.

It can be shown that this is due to the inefficient thread scheduling which involves unnecessary context switches. Consider a parallel network which consists of N process triples. These process triples are independent. This means, they can be executed one after another. Let t be the time a network consisting of a single process triple takes to execute. A schedule which executes one process triple after another would take $N \cdot t$ to execute.

This means, such a schedule would scale linearly with the number of process triples. In the current JavaFBP implementation, processes which are not connected do not block each other in a hidden way (e.g. by accessing a common resource). Thus, the worse than linear scaling (parallel, 10000 IPs) cannot be attributed to such an effect. The real execution only differs from the just described schedule in the fact that more than three processes are active at the same time, as was shown in section 6.3. This shows, that only the effects of the thread scheduling can be responsible for the worse scaling behavior.

6.5 Ideas For Improvement

In this section, some ideas for improving the current Java FBP implementation to overcome or at least mitigate the limits encountered during the conducted tests shall be presented.

6.5.1 Increasing Memory, Distribution

Section 6.3 showed that the number of processes allowed by JavaFBP is limited by available memory. The available testing environment had as much memory as possible on a 32-bit machine (4 GB). If more processes are needed in a network, a very obvious solution is deploying the application in a 64-bit environment on a machine with more memory.

Another solution would be distributing the network across multiple computers. As mentioned in section 2.3.1, this is included in the FBP concept and can be done by replacing in-memory connections with computer network connections probably employing sockets. This would increase the overall available memory as well as the parallelism.

A simple solution would be to introduce special components which handle the remote connections: Components which can read, and ones which can write to a socket, for example. A disadvantage of this solution is that the distribution gets visible in the network description. The externally communicating processes have to be explicitly declared and configured, and the network description has to be cut in pieces according to the distribution on multiple machines.

Furthermore, the remote connections are not visible to the FBP framework. This may prevent deadlocks from being detected. The first cause for this is that a process blocked on a remote connection still is in the “active” state. In section 2.4.4 is stated that a process being in the “active” state precludes a deadlock. This case would break this rule.

A second cause preventing deadlocks from being protected is that the deadlock detection logic can only see local processes. For successfully detecting a deadlock, the states

of all processes must be examined.

A clean solution would be to keep the distribution information separate from the network description. A network developer would first compose the network description and test some instances of it, and then a distribution plan would be created.

Such a distribution plan could be very simple: It could specify which process “lives” on which machine. From such a plan, the needed remote connections can be automatically determined using the following rule: If two processes which are directly connected are on different machines, the connection between them is remote, otherwise, if they are on the same machine, the connection is local.

A proposal how to implement such a distribution facility is beyond the scope of this work and leaves room for further research.

6.5.2 Cooperative M:N Scheduling

In section 6.4 it was shown that JavaFBP does not scale well when a network consists of many processes which are active at the same time. At least partially, this can be attributed to the fact that there are many unnecessary context switches done by the thread scheduler (see section 6.4.6). A context switch is considered unnecessary when a process, which could otherwise continue running, is interrupted to let another process get the CPU. To prevent these context switches and maximize the throughput, the scheduler could let the process run until it either blocks because of an empty or full connection, or terminates. The scheduler could have a list of processes waiting to get the CPU. As soon as a process yields the CPU because it cannot run anymore, another process from this waiting list gets the CPU and runs until it yields the CPU again, and so on. What the author has just described, is known as cooperative multitasking.

Some people may correctly argue that this is a step backwards because cooperative multitasking in its pure form does not make use of multiple CPUs. To overcome this limitation, the idea is to combine this with a pool of threads which can be scheduled by the operating system on the CPUs. Each thread would execute a process as long as possible and as soon as it blocks, the process is put on hold and the thread fetches a new one from the waiting list. Furthermore, when a process terminates, the thread discards it and fetches a new one from the list.

Essentially, this combines the scheduler of the operating system with a cooperative scheduler in user space. Scheduling M user space threads on N operating system threads is called M:N scheduling – hence the title of this section.

De Moura and Ierusalimsky [2004] show in their paper that cooperative multitasking can be implemented using coroutines (ibid., section 5.4). In this case, FBP processes would be implemented as coroutines. The blocking framework calls `OutputPort.send()` and `InputPort.receive()` would be implemented using the `yield` statement which would yield the control to the calling thread. If a process were ready to run again, the `resume` statement would be called which would resume the process where it left off.

Unfortunately, Java does not support coroutines. In the Java class library, a facility exists which can schedule tasks on threads using a thread pool. A task has to implement the `java.lang.Runnable` interface which contains a `run()` method. This method should contain the code to be concurrently executed. Such a task can be submitted to a `ThreadPoolExecutor` from the `java.util.concurrent` package. Such a `ThreadPoolExecutor` can be configured to have a set of threads which will obtain tasks from a queue and execute them.

One may have the idea to implement FBP processes as such tasks, but this does not work because it is not possible for a task to suspend itself and be resumed later, while the thread executes another task in between. The task has to finish completely before the thread can execute another one. In terms of FBP processes: A process would have to terminate before another one could be run by the executing thread.

Although Java does not support coroutines, there is the possibility to implement coroutines using continuations [Haynes et al., 1986]. The Java language does not include support for continuations, but there are some projects which offer continuation support for Java. The first one is “JavaFlow” (<http://commons.apache.org/sandbox/javaflow/>), a Apache Commons project which offers support for multi-shot continuations. A second one is “Kilim” (<http://www.malhar.net/sriram/kilim/>) which offers support for one-shot continuations.

What they have in common is that they both come with a library which offers some classes to work with continuations. Additionally, they provide a tool which does some modifications to the Java byte code to make continuations work (see for example Srinivasan [2006], section 2). One-shot continuations are less computationally expensive but also less powerful. But they suffice to implement coroutines [de Moura and Ierusalimsky, 2004, section 4.2]. Because they are faster than multi-shot continuations [Bruggerman et al., 1996], the author recommends the implementation provided by Kilim. The developer of the latter even does a performance comparison between Kilim and JavaFlow which indicates that the former is faster than the latter [Srinivasan, 2006, section 4].

In fact, Kilim implements a concept very similar to FBP. Processes are implemented by extending the `Task` class and get scheduled just as proposed above on a pool of threads. Connections are called mailboxes and behave very similar to FBP connections (FIFO order, many-to-one connections allowed) but they lack an end-of-data signal. Furthermore, testing for message availability on input ports is provided along with a facility to wait for messages becoming available on any input port and then do a receive on this port.

Perhaps merging the JavaFBP implementation with Kilim would be an interesting task for future work.

7 Conclusion

In chapter 2 the concepts of the Flow-Based Programming paradigm were portrayed along with advantages and problems. It was shown that FBP promotes loose coupling among components, allows for an efficient implementation by supporting concurrency very naturally and is very simple in its application. On the other hand, FBP networks may deadlock and errors in general are difficult to handle. While section 3 gave an overview, how FBP is related to other concepts that have appeared in the software architecture landscape, section 4 proposed some improvements which would make FBP less prone to deadlocks and thus more end user friendly.

How the Java implementation of FBP (JavaFBP) works was described in chapter 5. In particular, the deadlock handling was flawed in FBP. It shut down the whole application in which FBP was running. As this was not acceptable for large applications like PARASUITE, a new deadlock handling facility was created which shuts down the deadlocked FBP network and signals the error by throwing an exception.

Furthermore, a general facility to handle errors in FBP networks was missing. To remedy this, a simple error handling strategy was created which shuts down the whole FBP network in case of an error.

In chapter 6 the author examined how much processes an FBP network may consist of and found out that it solely depends on the amount of available memory. But depending on the topology and on the amount of data, the Java Virtual Machine runs out of different types of memory. In some cases, the amount of stack memory sets the barrier and in some cases the amount of heap memory does. Furthermore, if the amount of processes which have to be active at the same time is very high, the overall limit is reduced by several orders of magnitude.

Another finding is that JavaFBP does not scale well when several thousand processes are active at the same time. This is due to the scheduling overhead which is caused by unnecessary context switches.

In view of these results, some improvement proposals were made. The memory limit may be overcome by giving the application more memory or by distributing FBP networks across multiple computers. The scalability problems of JavaFBP can be remedied by limiting the number of threads and schedule the processes in user space on these threads in a cooperative manner.

7.1 Open Issues and Further Research

As FBP networks may potentially deadlock and end users are the network designers in PARASUITE, it may be interesting to develop facilities for signaling users that a certain network design might lead to a deadlock and dynamically provide some alternatives during design.

Another area of research would be to develop some other error handling strategies which might fit the term “graceful degradation” or go beyond that.

In regard to distributed FBP networks it would be interesting to examine distributed coordination, deadlock detection and error handling strategies.

Finally, when cooperative M:N scheduling as proposed in section 6.5.2 is realized, it might be of interest to run the tests performed in chapter 6 again and see whether there is still some room for improvement.

List of Figures

2.1	one-to-many connections are forbidden	4
2.2	many-to-one connections are allowed	5
2.3	FBP network using an instance of a subnet component	5
2.4	a FIFO queue	7
2.5	state chart – states of a process	7
2.6	processes blocked on receive	10
2.7	processes blocked on send	10
2.8	processes blocked on diverging streams	11
3.1	a SIFT process replacing itself consecutively with a FILTER and another SIFT process	13
4.1	Tagger processes inserting tags	15
5.1	class diagram of a hypothetical FBP framework	18
5.2	class diagram of the JavaFBP framework	19
5.3	the FIFO buffer	20
5.4	illustration of the method-call chain in case of an error	22
6.1	serial network topology	25
6.2	parallel network topology	27
6.3	Sierpinski triangles of different depths – 1 (top left), 2 (top right), 3 (bottom left), 5 (bottom right)	28
6.4	a two-layer triangle network	28
6.5	serial network, 1 IP, 12 to 10002 processes	35
6.6	serial network, 1 IP, 12 to 1002 processes	35
6.7	serial network, 1 IP, 12 to 102 processes	35
6.8	serial network, 10000 IPs, 12 to 5002 processes	36
6.9	serial network, 10000 IPs 12 to 1002 processes	37
6.10	serial network, 10000 IPs, 12 to 102 processes	37
6.11	parallel network, 1 IP, 30 to 600000 processes	39
6.12	parallel network, 1 IP, 30 to 300000 processes	39
6.13	parallel network, 1 IP, 30 to 30000 processes	39
6.14	parallel network, 1 IP, 30 to 3000 processes	40
6.15	parallel network, 1 IP, 30 to 300 processes	40
6.16	parallel network, 10000 IPs, 30 to 3000 processes	41
6.17	parallel network, 10000 IPs, 30 to 300 processes	42
6.18	triangle network, 1 IP, 3 to 3282 processes	43
6.19	triangle network, 1 IP, 3 to 1095 processes	44
6.20	triangle network, 1 IP, 3 to 366 processes	44
6.21	triangle network, 1 IP, 3 to 123 processes	44
6.22	triangle network, 10000 IPs, 3 to 366 processes	45
6.23	triangle network, 10000 IPs, 3 to 123 processes	45

Bibliography

- Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 99–107, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. doi: <http://doi.acm.org/10.1145/231379.231395>. URL <http://eprints.kfupm.edu.sa/62269/1/62269.pdf>. Retrieved July 12, 2009.
- Frank Buschmann, Regine Meunier, Hans Rohnert, Michael Stal, and Peter Sommerlad. *PATTERN-ORIENTED SOFTWARE ARCHITECTURE, A System of Patterns*. John Wiley & Sons Ltd, 1996. ISBN 0-471-95869-7.
- Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. 2004. URL <http://www.inf.puc-rio.br/~roberto/docs/MCC15-04.pdf>. Retrieved July 11, 2009.
- Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/1297105.1297033>. also available at <http://buytaert.net/files/oopsla07-georges.pdf>, retrieved July 9, 2009.
- Christopher T Haynes, Daniel P Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Comput. Lang.*, 11(3-4):143–153, 1986. ISSN 0096-0551. doi: [http://dx.doi.org/10.1016/0096-0551\(86\)90007-X](http://dx.doi.org/10.1016/0096-0551(86)90007-X).
- Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. 1977.
- R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. pages 483–499, 1996. Also available from <http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf>, retrieved July 13, 2009.
- Henry Lieberman, Fabio Parternò, Markus Klann, and Volker Wulf. End-user development: An emerging paradigm. *End User Development*, 9:1–8, 2006. ISSN 1571-5035. doi: 10.1007/1-4020-5386-X.
- Sun Microsystems. *Java™ Platform, Standard Edition 6 API Specification*, a. URL <http://java.sun.com/javase/6/docs/api/>. Retrieved July 6, 2009.
- Sun Microsystems. *Server-Class Machine Detection*, b. URL <http://java.sun.com/javase/6/docs/technotes/guides/vm/server-class.html>. Retrieved June 23, 2009.
- John Paul Morrison. *Flow-Based Programming*. van Nostrand Reinhold, New York, 1994. ISBN 0-442-01771-5. URL <http://www.jpaulmorrison.com/fbp/book.pdf>.
- John Paul Morrison. Flowbasedprogramming. URL <http://www.jpaulmorrison.com/cgi-bin/wiki.pl>. Retrieved June 23, 2009.
- Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, New Jersey, 1996. ISBN 0-13-182957-2.

Sriram Srinivasan. A thread of one's own. 2006. URL http://www.malhar.net/sriram/kilim/thread_of_ones_own.pdf. Retrieved July12, 2009.

Wayne P. Stevens. *Software design: concepts and methods*. Prentice Hall International (UK) Ltd, Hertfordshire, 1991. ISBN 0-13-820242-7.

Edward Yourdon and Larry L. Constantine. *Structured design*. Prentice-Hall, New Jersey, 1979. ISBN 0-13-854471-9.

A Source Code

A.1 JavaStats

A.1.1 javastats.conf

```
1  [general]
2  benchmark suites: serial parallel triangle
3  virtual machines: sun
4  output file: results
5
6  [trace]
7  machine: localhost
8  location: /tmp
9  prefix: javastats_trace
10
11 [performance]
12 class: TimePerformance
13
14 [stats]
15 minimum vm invocations: 10
16 maximum vm invocations: 50
17 minimum benchmark iterations: 1
18 maximum benchmark iterations: 1
19 confidence level: 0.95
20 stop criterium: percentage
21 stop threshold: 2.0
22
23 [sun]
24 binary location: /usr/bin
25 binary name: java
26
27 [serial]
28 location: /home/master/Masterarbeit
29 input sizes: 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 1000 2000 3000 4000
30              5000 6000 7000 8000 9000 10000
31 startup command: -XX:+UseParallelGC -server -jar Test.jar serial %(benchmark)s %(input)s 10
32 steady command: -XX:+UseParallelGC -server -jar Test.jar serial %(benchmark)s %(input)s 10
33 benchmarks: 1 10000
34 ulimit threshold:
35
36 [parallel]
37 location: /home/master/Masterarbeit
38 input sizes: 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 1000 2000 3000 4000
39              5000 6000 7000 8000 9000 10000 20000 30000 40000 50000 60000 70000 80000 90000
40              100000 200000
41 startup command: -XX:+UseParallelGC -server -jar Test.jar parallel %(benchmark)s %(input)s 10
42 steady command: -XX:+UseParallelGC -server -jar Test.jar parallel %(benchmark)s %(input)s 10
43 benchmarks: 1 10000
44 ulimit threshold:
45
46 [triangle]
47 location: /home/master/Masterarbeit
48 input sizes: 1 2 3 4 5 6 7 8 9 10
49 startup command: -XX:+UseParallelGC -server -jar Test.jar triangle %(benchmark)s %(input)s 10
50 steady command: -XX:+UseParallelGC -server -jar Test.jar triangle %(benchmark)s %(input)s 10
51 benchmarks: 1
52 ulimit threshold:
53
```

A.1.2 performance.py

```
1  #!/usr/bin/env python
2
3  # JavaStats is a toolkit designed to get a
4  # statistically rigorous performance evaluation for a
5  # given (Java) application
6  #
```

```

7 # Copyright (C) 2007 Andy Georges
8 #
9 # This program is free software; you can redistribute it and/or
10 # modify it under the terms of the GNU General Public License
11 # as published by the Free Software Foundation; either version 2
12 # of the License, or (at your option) any later version.
13 #
14 # This program is distributed in the hope that it will be useful,
15 # but WITHOUT ANY WARRANTY; without even the implied warranty of
16 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
17 # GNU General Public License for more details.
18 #
19 # You should have received a copy of the GNU General Public License
20 # along with this program; if not, write to the Free Software
21 # Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
22
23 # This file implements several useful example snippets that can be used to
24 # return a desired performance number to the main JavaStats monitor. Basically,
25 # you need to implement two methods given in the Performance base class:
26 # acquire_command and get_performance_data. The former will adjust the java
27 # run command and set it up to gather the desired data. The latter will examine
28 # the resulting trace file and fetch the desired data.
29
30 import re
31
32 class Performance:
33     """Performance is the base class for determining the performance
34     according to some criterion (e.g., time, executed instruction,
35     etc.) of a Java execution.
36     """
37     # regular expression that check if an error occured during a run
38     # as well as the regex for the runtime in startup mode
39     re_exception = re.compile("[Ee]xception")
40     re_error = re.compile("Error")
41     re_deadlock = re.compile("deadlock")
42     re_terminated = re.compile("terminated_by")
43     re_non_zero = re.compile("non_zero_status")
44     re_stack = re.compile("__Stack__")
45     re_failed = re.compile("FAILED")
46     re_not_valid = re.compile("NOT_VALID")
47
48     def acquire_command(self, commmand):
49         pass
50
51     def get_performance_data(self, trace_filename):
52         pass
53
54     def sweep_line_for_error(self, line):
55         """The sweep_line_for_error function will try to find selected regular
56         expressions in the line that indicate an error or unexpected situation
57         occurred during the execution. If that is the case, the experiment
58         should be discarded.
59         """
60         if self.re_exception.search(line):
61             return True
62         if self.re_error.search(line):
63             return True
64         if self.re_deadlock.search(line):
65             return True
66         if self.re_terminated.search(line):
67             return True
68         if self.re_non_zero.search(line):
69             return True
70         if self.re_stack.search(line):
71             return True
72         if self.re_failed.search(line):
73             return True
74         if self.re_not_valid.search(line):
75             return True
76
77         return False
78
79
80 class TimePerformance(Performance):
81
82     re_time = re.compile("[0-9]+_[0-9]+_[0-9]+_[0-9]+")

```

```

83
84 def acquire_command(self, java_command):
85     return java_command
86
87 def get_performance_data(self, trace_filename):
88
89     f = file(trace_filename, 'r')
90     for line in f.readlines():
91         if self.sweep_line_for_error(line):
92             return None
93         if self.re_time.search(line):
94             return float(line.strip().split("_")[-1])

```

A.2 Test Program

A.2.1 Class run.Main

```

1 package run;
2
3 import org.apache.commons.math.stat.descriptive.DescriptiveStatistics;
4
5 import networks.ParallelNetwork;
6 import networks.SerialNetwork;
7 import networks.TestNetwork;
8 import networks.TriangleNetwork;
9
10 public class Main {
11     /**
12      * @param args
13      * @throws Exception
14      */
15     public static void main(String [] args) throws Exception {
16         if (args.length < 3) {
17             System.out.println("Usage: _Main_<topology>_<number_of_packets>_"
18                 + "<network_size>_(<connection_capacity>)");
19             System.exit(0);
20         }
21
22         int numberOfPackets = Integer.parseInt(args[1]);
23         int networkSize = Integer.parseInt(args[2]);
24         int connectionCapacity = 10;
25         if (args.length == 4) {
26             connectionCapacity = Integer.parseInt(args[3]);
27         }
28         boolean warmup = true;
29         boolean singleIteration = false;
30         boolean onlyDefine = false;
31         if (args.length == 5) {
32             if (args[4].contains("nowarmup")) {
33                 warmup = false;
34             }
35             if (args[4].contains("single")) {
36                 singleIteration = true;
37             }
38             if (args[4].contains("onlydefine")) {
39                 onlyDefine = true;
40             }
41         }
42
43         String networkType = args[0];
44         TestNetwork network = determineNetwork(networkType, numberOfPackets,
45             networkSize, connectionCapacity);
46
47         if (network != null) {
48             // do warm up if requested
49             if (warmup) {
50                 int runs = 1;
51                 DescriptiveStatistics stat = new DescriptiveStatistics(5);
52                 long startWarmup = System.currentTimeMillis();
53                 network.warmUp();
54                 stat.addValue(System.currentTimeMillis() - startWarmup);
55                 double cov;
56                 do {
57                     startWarmup = System.currentTimeMillis();
58                     network.warmUp();
59                     stat.addValue(System.currentTimeMillis() - startWarmup);

```

```

60         runs++;
61         cov = stat.getStandardDeviation() / stat.getMean();
62         if (cov < 0.03) {
63             System.out.println("CoV:_" + cov);
64         }
65     } while (cov > 0.02);
66     System.out.println("runs:_" + runs);
67 }
68 long sumRunDuration = 0;
69 long sumDefineDuration = 0;
70 long numberOfComponents = 0;
71 long numberOfConnections = 0;
72 if (singleIteration) {
73     // do only a single iteration
74     System.out.println("start_define");
75     System.out.flush();
76     long startDefine = System.currentTimeMillis();
77     network.callDefine();
78     long defineDuration = System.currentTimeMillis() - startDefine;
79     numberOfComponents = network.getNumberOfComponents();
80     numberOfConnections = network.getNumberOfConnections();
81     System.out.println("define_ready");
82     System.out.flush();
83     long runDuration = 0;
84     if (!onlyDefine) {
85         Thread.sleep(1000);
86         long startRun = System.currentTimeMillis();
87         network.doRun();
88         runDuration = System.currentTimeMillis() - startRun;
89     }
90     System.out.print(numberOfComponents + "_" + numberOfConnections
91         + "_");
92     System.out.println(defineDuration + "_" + runDuration);
93 } else {
94     // do five iterations and calculate the mean
95     for (int i = 0; i < 5; i++) {
96         System.out.println("start_define");
97         System.out.flush();
98         if (i > 0) {
99             network = determineNetwork(networkType,
100                 numberOfPackets, networkSize,
101                 connectionCapacity);
102         }
103         long startDefine = System.currentTimeMillis();
104         network.callDefine();
105         sumDefineDuration += System.currentTimeMillis()
106             - startDefine;
107         if (i == 0) {
108             numberOfComponents = network.getNumberOfComponents();
109             numberOfConnections = network.getNumberOfConnections();
110         }
111         System.out.println("define_ready");
112         System.out.flush();
113         // Thread.sleep(1000);
114         if (!onlyDefine) {
115             long startRun = System.currentTimeMillis();
116             network.doRun();
117             sumRunDuration += System.currentTimeMillis() - startRun;
118         }
119     }
120
121     long defineDuration = Math.round(sumDefineDuration / 5.0);
122     long runDuration = Math.round(sumRunDuration / 5.0);
123     System.out.print(numberOfComponents + "_" + numberOfConnections
124         + "_");
125     System.out.println(defineDuration + "_" + runDuration);
126 }
127 }
128 }
129
130 private static TestNetwork determineNetwork(String networkType,
131     int numberOfPackets, int networkSize, int connectionCapacity) {
132     // create a network as specified on the command line
133     if (networkType.equalsIgnoreCase("serial")) {
134         return new SerialNetwork(numberOfPackets, networkSize,
135             connectionCapacity);

```

```

136         } else if (networkType.equalsIgnoreCase("parallel")) {
137             return new ParallelNetwork(numberOfPackets, networkSize,
138                 connectionCapacity);
139         } else if (networkType.equalsIgnoreCase("triangle")) {
140             return new TriangleNetwork(numberOfPackets, networkSize,
141                 connectionCapacity);
142         }
143         return null;
144     }
145 }

```

A.2.2 Class networks.TestNetwork

```

1 package networks;
2
3 import com.jp Morrnsn.fbp.engine.Network;
4
5 public abstract class TestNetwork extends Network {
6     public abstract int getNumberOfConnections();
7     public abstract void warmUp() throws Exception;
8 }

```

A.2.3 Class networks.SerialNetwork

```

1 package networks;
2
3 import com.jp Morrnsn.fbp.components.Discard;
4 import com.jp Morrnsn.fbp.engine.Component;
5 import com.jp Morrnsn.fbp.engine.Port;
6
7 import components.Generate;
8 import components.Square;
9
10 public class SerialNetwork extends TestNetwork {
11
12     private int numberOfComponents;
13     private int numberOfPackets;
14     private int connectionCapacity;
15     private int connCount = 0;
16
17     public SerialNetwork(int numberOfPackets, int numberOfComponents,
18         int connectionCapacity) {
19         this.numberOfPackets = numberOfPackets;
20         this.numberOfComponents = numberOfComponents;
21         this.connectionCapacity = connectionCapacity;
22         traceLocks = false;
23         tracing = false;
24     }
25
26     @Override
27     protected void define() throws Exception {
28         Component lastComponent = component("generate", Generate.class);
29         component("discard", Discard.class);
30
31         for (int i = 0; i < numberOfComponents; i++) {
32             Component newComponent = component("inner" + i, Square.class);
33             conn(lastComponent, port("OUT"), newComponent, port("IN"),
34                 connectionCapacity);
35             lastComponent = newComponent;
36         }
37         conn(lastComponent, port("OUT"), "discard.IN", connectionCapacity);
38
39         initialize(numberOfPackets, "generate.PACKETS");
40     }
41
42     private void conn(Component sender, Port port, String receiver,
43         int connectionCapacity) {
44         connCount++;
45         connect(sender, port, receiver, connectionCapacity);
46     }
47
48     private void conn(Component sender, Port sourcePort,
49         Component receiver, Port destinationPort, int connectionCapacity) {
50         connCount++;
51         connect(sender, sourcePort, receiver, destinationPort, connectionCapacity);
52     }

```

```

53
54     @Override
55     public int getNumberOfConnections() {
56         return connCount;
57     }
58
59     @Override
60     public void warmUp() throws Exception {
61         // run a small serial network in a warm-up phase (10 IPs, 102 processes,
62         // connection size of 10)
63         SerialNetwork net = new SerialNetwork(10, 100, 10);
64         net.callDefine();
65         net.doRun();
66     }
67 }

```

A.2.4 Class networks.ParallelNetwork

```

1  package networks;
2
3  import com.jp Morrnsn.fbp.components.Discard;
4  import com.jp Morrnsn.fbp.engine.Component;
5  import com.jp Morrnsn.fbp.engine.Port;
6
7  import components.Generate;
8  import components.Square;
9
10 public class ParallelNetwork extends TestNetwork {
11
12     private int numberOfPackets;
13     private int numberOfComponents;
14     private int connectionCapacity;
15     private int connCount = 0;
16
17     public ParallelNetwork(int numberOfPackets, int numberOfComponents,
18         int connectionCapacity) {
19         this.numberOfPackets = numberOfPackets;
20         this.numberOfComponents = numberOfComponents;
21         this.connectionCapacity = connectionCapacity;
22         tracing = false;
23         traceLocks = false;
24     }
25
26     @Override
27     protected void define() throws Exception {
28         for (int i = 0; i < numberOfComponents; i++) {
29             Component generate = component("generate" + i, Generate.class);
30             Component innerComponent = component("inner" + i, Square.class);
31             Component discard = component("discard" + i, Discard.class);
32             conn(generate, port("OUT"), innerComponent, port("IN"), connectionCapacity);
33             conn(innerComponent, port("OUT"), discard, port("IN"), connectionCapacity);
34             initialize(numberOfPackets, generate, port("PACKETS"));
35         }
36     }
37
38     private void conn(Component sender, Port sourcePort, Component receiver,
39         Port destinationPort, int connectionCapacity) {
40         connCount++;
41         connect(sender, sourcePort, receiver, destinationPort, connectionCapacity);
42     }
43
44     @Override
45     public int getNumberOfConnections() {
46         return connCount;
47     }
48
49     @Override
50     public void warmUp() throws Exception {
51         // run a small parallel network in a warm-up phase (10 IPs, 999 processes,
52         // connection size of 10)
53         ParallelNetwork net = new ParallelNetwork(10, 333, 10);
54         net.callDefine();
55         net.doRun();
56     }
57 }

```

A.2.5 Class networks.TriangleNetwork

```

1  package networks;
2
3  import com.jp Morrnsn.fbp.components.Discard;
4  import com.jp Morrnsn.fbp.engine.Component;
5  import com.jp Morrnsn.fbp.engine.Port;
6
7  import components.SierpinskiSelector;
8  import components.TriangleGenerator;
9
10 public class TriangleNetwork extends TestNetwork {
11
12     private int numberOfPackets;
13     private int numberOfComponents;
14     private int id = 0;
15     private int connCount = 0;
16     private int connectionCapacity;
17
18     public TriangleNetwork(int numberOfPackets, int numberOfComponents,
19         int connectionCapacity) {
20         this.numberOfPackets = numberOfPackets;
21         this.numberOfComponents = numberOfComponents;
22         this.connectionCapacity = connectionCapacity;
23         traceLocks = false;
24         tracing = false;
25         deadlockTest = false; // disable deadlock testing because of a bug in
26                               // JavaFBP
27     }
28
29     @Override
30     protected void define() throws Exception {
31         Component generator = component("generator", TriangleGenerator.class);
32         Component evaluator = component("evaluator", Discard.class);
33         Component selector = component("selector", SierpinskiSelector.class);
34         conn(generator, port("OUT"), selector, port("IN"), connectionCapacity);
35         createNetwork(numberOfComponents - 1, selector, evaluator);
36         initialize(numberOfPackets, generator, port("TRIANGLES"));
37     }
38
39     private void createNetwork(int depth, Component generator,
40         Component evaluator) {
41         if (depth < 1) {
42             conn(generator, port("LEFT"), evaluator, port("IN"),
43                 connectionCapacity);
44             conn(generator, port("RIGHT"), evaluator, port("IN"),
45                 connectionCapacity);
46             conn(generator, port("TOP"), evaluator, port("IN"),
47                 connectionCapacity);
48         } else {
49             id++;
50             Component left = component("left" + id, SierpinskiSelector.class);
51             Component right = component("right" + id, SierpinskiSelector.class);
52             Component top = component("top" + id, SierpinskiSelector.class);
53             conn(generator, port("LEFT"), left, port("IN"), connectionCapacity);
54             conn(generator, port("RIGHT"), right, port("IN"), connectionCapacity);
55             conn(generator, port("TOP"), top, port("IN"), connectionCapacity);
56             createNetwork(depth - 1, left, evaluator);
57             createNetwork(depth - 1, right, evaluator);
58             createNetwork(depth - 1, top, evaluator);
59         }
60     }
61
62     private void conn(Component sc, Port sp, Component dc, Port dp, int size) {
63         connect(sc, sp, dc, dp, size);
64         connCount++;
65     }
66
67     @Override
68     public int getNumberOfConnections() {
69         return connCount;
70     }
71
72     @Override
73     public void warmUp() throws Exception {
74         // run a small triangle network in a warm-up phase (10 IPs, 366 processes,

```

```

75         // connection size of 10)
76         TriangleNetwork net = new TriangleNetwork(10, 6, 10);
77         net.callDefine();
78         net.doRun();
79     }
80 }

```

A.2.6 Class datastructures.Triangle

```

1  package datastructures;
2
3  public class Triangle implements Cloneable {
4      private double[] a;
5      private double[] b;
6      private double[] c;
7      private int id;
8
9      public Triangle(int id, double ax, double ay, double bx, double by,
10         double cx, double cy) {
11         this.id = id;
12         double[] aTemp = {ax, ay};
13         a = aTemp;
14         double[] bTemp = {bx, by};
15         b = bTemp;
16         double[] cTemp = {cx, cy};
17         c = cTemp;
18     }
19
20     public double[] getA() {
21         return a;
22     }
23
24     public double[] getB() {
25         return b;
26     }
27
28     public double[] getC() {
29         return c;
30     }
31
32     public int getId() {
33         return id;
34     }
35 }

```

A.2.7 Class components.Generate

```

1  package components;
2
3  import com.jp Morrnsn.fbp.engine.Component;
4  import com.jp Morrnsn.fbp.engine.InPort;
5  import com.jp Morrnsn.fbp.engine.InputPort;
6  import com.jp Morrnsn.fbp.engine.OutputPort;
7  import com.jp Morrnsn.fbp.engine.OutputPort;
8  import com.jp Morrnsn.fbp.engine.Packet;
9
10 @InPort("PACKETS")
11 @OutPort("OUT")
12 public class Generate extends Component {
13
14     private InputPort packetsPort;
15     private OutputPort outputPort;
16
17     @Override
18     protected void execute() throws Exception {
19         Packet conf = packetsPort.receive();
20         int numberOfPackets = (Integer) conf.getContent();
21         drop(conf);
22         packetsPort.close();
23
24         for (int i = 0; i < numberOfPackets; i++) {
25             outputPort.send(create(i));
26         }
27     }
28
29     @Override

```

```

30     protected void openPorts() {
31         packetsPort = openInput("PACKETS");
32         outputPort = openOutput("OUT");
33     }
34 }
35 }

```

A.2.8 Class components.SierpinskiSelector

```

1  package components;
2
3  import com.jpMorrsn.fbp.engine.Component;
4  import com.jpMorrsn.fbp.engine.InPort;
5  import com.jpMorrsn.fbp.engine.InputPort;
6  import com.jpMorrsn.fbp.engine.OutputPort;
7  import com.jpMorrsn.fbp.engine.OutputPorts;
8  import com.jpMorrsn.fbp.engine.OutputPort;
9  import com.jpMorrsn.fbp.engine.Packet;
10
11 import datastructures.Triangle;
12
13 @InPort("IN")
14 @OutPorts({
15     @OutPort("LEFT"),
16     @OutPort("RIGHT"),
17     @OutPort("TOP")
18 })
19 public class SierpinskiSelector extends Component {
20     private InputPort inputPort;
21     private OutputPort leftPort;
22     private OutputPort rightPort;
23     private OutputPort topPort;
24
25     @Override
26     protected void execute() throws Exception {
27         Packet p;
28         while ((p = inputPort.receive()) != null) {
29             Triangle triangle = (Triangle) p.getContent();
30             drop(p);
31             sendLeft(triangle);
32             sendRight(triangle);
33             sendTop(triangle);
34         }
35     }
36
37     private void sendTop(Triangle triangle) {
38         int id = triangle.getId();
39         double[] a = triangle.getA();
40         double[] b = triangle.getB();
41         double[] c = triangle.getC();
42         double ax = c[0] + (a[0] - c[0]) / 2.0;
43         double ay = c[1] + (a[1] - c[1]) / 2.0;
44         double bx = c[0] + (b[0] - c[0]) / 2.0;
45         double by = c[1] + (b[1] - c[1]) / 2.0;
46         topPort.send(create(new Triangle(id, ax, ay, bx, by, c[0], c[1])));
47     }
48
49     private void sendRight(Triangle triangle) {
50         int id = triangle.getId();
51         double[] a = triangle.getA();
52         double[] b = triangle.getB();
53         double[] c = triangle.getC();
54         double ax = b[0] + (a[0] - b[0]) / 2.0;
55         double ay = b[1] + (a[1] - b[1]) / 2.0;
56         double cx = b[0] + (c[0] - b[0]) / 2.0;
57         double cy = b[1] + (c[1] - b[1]) / 2.0;
58         rightPort.send(create(new Triangle(id, ax, ay, b[0], b[1], cx, cy)));
59     }
60
61     private void sendLeft(Triangle triangle) {
62         int id = triangle.getId();
63         double[] a = triangle.getA();
64         double[] b = triangle.getB();
65         double[] c = triangle.getC();
66         double bx = a[0] + (b[0] - a[0]) / 2.0;
67         double by = a[1] + (b[1] - a[1]) / 2.0;

```

```

68         double cx = a[0] + (c[0] - a[0]) / 2.0;
69         double cy = a[1] + (c[1] - a[1]) / 2.0;
70         leftPort.send(create(new Triangle(id, a[0], a[1], bx, by, cx, cy)));
71     }
72
73     @Override
74     protected void openPorts() {
75         inputPort = openInput("IN");
76         leftPort = openOutput("LEFT");
77         rightPort = openOutput("RIGHT");
78         topPort = openOutput("TOP");
79     }
80
81 }

```

A.2.9 Class components.Square

```

1  package components;
2
3  import com.jpmorrisn.fbp.engine.Component;
4  import com.jpmorrisn.fbp.engine.InPort;
5  import com.jpmorrisn.fbp.engine.InputPort;
6  import com.jpmorrisn.fbp.engine.OutputPort;
7  import com.jpmorrisn.fbp.engine.OutputPort;
8  import com.jpmorrisn.fbp.engine.Packet;
9
10 @InPort("IN")
11 @OutPort("OUT")
12 public class Square extends Component {
13
14     private InputPort inputPort;
15     private OutputPort outputPort;
16
17     @Override
18     protected void execute() throws Exception {
19         Packet p;
20         while ((p = inputPort.receive()) != null) {
21             int i = (Integer) p.getContent();
22             drop(p);
23             outputPort.send(create(i*i));
24         }
25     }
26
27     @Override
28     protected void openPorts() {
29         inputPort = openInput("IN");
30         outputPort = openOutput("OUT");
31     }
32
33 }

```

A.2.10 Class components.TriangleGenerator

```

1  package components;
2
3  import java.util.Random;
4
5  import com.jpmorrisn.fbp.engine.Component;
6  import com.jpmorrisn.fbp.engine.InPort;
7  import com.jpmorrisn.fbp.engine.InputPort;
8  import com.jpmorrisn.fbp.engine.OutputPort;
9  import com.jpmorrisn.fbp.engine.OutputPort;
10 import com.jpmorrisn.fbp.engine.Packet;
11
12 import datastructures.Triangle;
13
14 @InPort("TRIANGLES")
15 @OutPort("OUT")
16 public class TriangleGenerator extends Component {
17
18     private InputPort trianglesPort;
19     private OutputPort outputPort;
20
21     @Override
22     protected void execute() throws Exception {
23         Packet cp = trianglesPort.receive();

```

A Source Code

```
24     trianglesPort.close();
25     int numberOfTriangles = (Integer) cp.getContent();
26     drop(cp);
27
28     Random random = new Random();
29     for (int i = 0; i < numberOfTriangles; i++) {
30         double length = Math.abs(8.0 + 2.0 * random.nextGaussian());
31         double half = length / 2.0;
32         Triangle triangle = new Triangle(i, 0.0, 0.0,
33             length, 0.0,
34             half, Math.sqrt(length * length - half * half));
35         outputPort.send(create(triangle));
36     }
37 }
38
39 @Override
40 protected void openPorts() {
41     trianglesPort = openInput("TRIANGLES");
42     outputPort = openOutput("OUT");
43 }
44
45 }
```