Evaluation And Extension Of an Implementation Of Flow-Based Programming

Sven Steinseifer

Referent: Prof. Letschert
Koreferent: Prof. Geisse

July 12, 2009
## What Is Flow-Based Programming?

1. **History** .......................... 5
2. **FBP – a Programming Paradigm** .......................... 5
   2.1 The FBP Language .......................... 5
   2.2 Hierarchical Decomposition .......................... 6
   2.3 Data Grouping .......................... 6
3. **Component Model** .......................... 7
   3.1 Components and Processes .......................... 7
   3.2 Asynchronous Message Passing .......................... 7
   3.3 Scheduling .......................... 8
   3.4 Subnets and Substreams .......................... 8
4. **Properties of FBP** .......................... 9
   4.1 Loose Coupling .......................... 9
   4.2 Efficiency .......................... 9
   4.3 Simplicity .......................... 10
   4.4 Problems and Challenges .......................... 10

## Java Implementation of FBP

1. **Usage of the JavaFBP Framework** .......................... 11
   1.1 The FBP Language of JavaFBP .......................... 11
   1.2 Implementing Components .......................... 11
   1.3 Creating Subnets .......................... 11
2. **How does JavaFBP implement FBP?** .......................... 12
   2.1 Statical Design – Important Classes .......................... 12
   2.2 Dynamics – Which Class Does What? .......................... 12
   2.3 Flaws, Inconveniences and Fixes .......................... 13

## Performance and Scalability

1. **The Testing Environment** .......................... 16
2. **Network Topologies** .......................... 17
   2.1 The Serial Topology .......................... 17
   2.2 The Parallel Topology .......................... 18
   2.3 The Triangle Topology .......................... 18
   2.4 Rationale .......................... 19
   2.5 Test Program .......................... 19
3. **How many processes does the JRE allow?** .......................... 20
4. **How does JavaFBP scale?** .......................... 23
   4.1 Statistically Rigorous Performance Evaluation .......................... 23
   4.2 Tools .......................... 24
   4.3 Serial Networks .......................... 24
   4.4 Parallel Networks .......................... 28
   4.5 Triangle Networks .......................... 31
5. **Ideas For Improvement** .......................... 32
   5.1 Increasing Memory, Distribution .......................... 32
   5.2 Cooperative M:N Scheduling .......................... 34
A Source Code

A.1 JavaStats

A.1.1 javastats.conf

A.1.2 performance.py
1 What Is Flow-Based Programming?

1.1 History

“Flow-Based Programming” (short: FBP) 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, 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.

1.2 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.

1.2.1 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.
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 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 one.

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. 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 or leaving it unconnected.

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.

1.2.2 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.

1.2.3 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. (TODO: siehe später)

\[1\] In 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 2.1.1.
1.3 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.

1.3.1 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 1.2, 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 provides the application programming interface (short: 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).

1.3.2 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 (short: FIFO) order. 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.

A message is called “Information Packet” (short: 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.

Another feature of IPs is that they cannot be lost. As soon as an IPs 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 1.3.3), 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 1.2.1 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” (short: 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.
1.3.3 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

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 is also 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.

1.3.4 Subnets and Substreams

The data grouping facility mentioned in section 1.2.3 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 sub-
net. Output ports of the subnet may be configured “substream sensitive”, too. These
will add the bracket IPs which were removed at the input ports to keep the grouping
intact.

1.4 Properties of FBP

1.4.1 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 com-
munication between components. Coupling which results from using shared memory
between components – called “common-environment coupling” in [Yourdon and Con-
stantine 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 com-
ponent 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 that 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 bind-
ing 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 be-
cause 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 prom-
otes reuse [Buschmann et al. 1996, p. 68; Shaw and Garlan 1996, p. 22]. Third, it
eases maintainability: “...old filters can replaced by improved ones” [Shaw and Gar-
lan 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).

1.4.2 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 components as
filters and connections as pipes. According to Shaw and Garlan 1996 such systems “nat-
urally support concurrent execution” (ibid., p. 22). Especially on multi-processor ma-
chines 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.

1.4.3 Simplicity
As already stated in section 1.2.1 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 1.3.2).

1.4.4 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 circle block on receiving from empty channels. I.e., they expect each respective predecessor to do a send.
2. All processes involved in the circle block on sending to full channels. I.e., they expect each respective successor to do a receive.

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.

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].

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 could be implemented is explained in section 2.2.3.

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

2.1 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 documentation.

2.1.1 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 1.2.1 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.

2.1.2 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.

2.1.3 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

1 can 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 a FBP network.

2.2 How does JavaFBP implement FBP?

2.2.1 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.

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).

The JavaFBP developers followed a similar approach but opted, however, for a single-inheritance class hierarchy. 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.

2.2.2 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 1.3.3)
- 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 2.1.2 the execute() method contains the component’s logic. When the process gets started, this method is invoked. According to the scheduling rules (section 1.3.3), 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.

To 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.
In section 1.3.2 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. 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 the empty channel, the wait() method inherited from the java.lang.Object class is invoked to release the lock and give sending 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 1.4.4: 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.

2.2.3 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 (short: 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
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 are not 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 an 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 1.4.4 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 \texttt{execute()} method would return a special error code to the framework [Morrison, 1994, p. 122]. According to the Java conventions for signaling errors, the \texttt{execute()} method should throw an exception in case of an error. Because the \texttt{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 \texttt{throws} declaration. The choice fell on the \texttt{java.lang.Exception} class. Most “normal” exceptions inherit from this class. A premature choice would have been to use the \texttt{java.lang.Throwable} base class. But this would have included, in addition to the exceptions, classes inheriting from \texttt{java.lang.Error}. Errors, in the sense of inheriting from this class, are mostly abnormal conditions like out of memory errors which cannot reasonably 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 \texttt{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 it 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 \texttt{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 \texttt{go()} method and calls its children’s \texttt{terminate()} method which is responsible for terminating the called node. The \texttt{terminate()} method of subnet calls the respective methods of its direct child nodes.

Java provides a method (\texttt{Thread.stop()}) to immediately stop threads. But this method is deprecated due to various reasons (see the documentation in [Microsystems]). As with handling deadlocks, the \texttt{interrupt()} method is used instead. The \texttt{OutputPort.send()} and \texttt{InputPort.receive()} methods will react eventually throwing an \texttt{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 \texttt{interrupt} flag with the \texttt{interrupted()} or \texttt{isInterrupted()} of the \texttt{Thread} class during lengthy calculations and let the process terminate.
3 Performance and Scalability

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.

3.1 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 (short: 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 Microsystems[4], 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 (short: 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.

3.2 Network Topologies

3.2.1 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. The data source component is called Generate and implemented as follows:

```java
package components;

import com.jpmorra.fbp.engine.Component;
import com.jpmorra.fbp.engine.InPort;
import com.jpmorra.fbp.engine.InputPort;
import com.jpmorra.fbp.engine.OutPort;
import com.jpmorra.fbp.engine.OutputPort;
import com.jpmorra.fbp.engine.Packet;

@InPort("PACKETS")
@OutPort("OUT")
public class Generate extends Component {
    private InputPort packetsPort;
    private OutputPort outputPort;

    @Override
    protected void execute() throws Exception {
        Packet conf = packetsPort.receive();
        int numberOfPackets = (Integer) conf.getContent();
        drop(conf);
        packetsPort.close();

        for (int i = 0; i < numberOfPackets; i++) {
            outputPort.send(create(i));
        }
    }

    @Override
    protected void openPorts() {
        packetsPort = openInput("PACKETS");
        outputPort = openOutput("OUT");
    }
}
```
This component does the following: It reads the number of IPs (n) to be produced from the PACKETS port (lines 19 and 20) and outputs n IPs containing the values 0, 1, \ldots, n-1 (lines 24-26).

The component used for calculating processes is called \texttt{Square} and looks like this:

```java
package components;

import com.jpmorrsn.fbp.engine.Component;
import com.jpmorrsn.fbp.engine.InPort;
import com.jpmorrsn.fbp.engine.InputPort;
import com.jpmorrsn.fbp.engine.OutPort;
import com.jpmorrsn.fbp.engine.OutputPort;
import com.jpmorrsn.fbp.engine.Packet;

@InPort("IN")
@OutPort("OUT")
public class Square extends Component {
    private InputPort inputPort;
    private OutputPort outputPort;

    @Override
    protected void execute() throws Exception {
        Packet p;
        while ((p = inputPort.receive()) != null) {
            int i = (Integer) p.getContent();
            drop(p);
            outputPort.send(create(i*i));
        }
    }

    @Override
    protected void openPorts() {
        inputPort = openInput("IN");
        outputPort = openOutput("OUT");
    }
}
```

This component reads IPs from the \texttt{IN} port (lines 18 and 19) and sends the square of each IP’s value to the \texttt{OUT} port (line 21).

The data sink component is the \texttt{Discard} component as shipped with the JavaFBP framework. This component simply drops each incoming IP.

### 3.2.2 The Parallel Topology

The second network topology is the so-called “parallel” topology. Networks of this topology consist of a configurable number of small, independent networks with a \texttt{Generate} process connected to a \texttt{Square} process which, finally, is connected to a \texttt{Discard} process.

### 3.2.3 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
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 triangle.
3. Remove the resulting triangle.
4. For each of the three resulting triangles, start again at step 2.

Networks with the triangle topology consist of a data source called TriangleGenerator which generates a configurable amount of IPs containing triangle descriptions. These descriptions consist of the coordinates 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 for each output port 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.

3.2.4 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.

3.2.5 Test Program

To conduct the following tests, a test program 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 \texttt{nowarmup}, no warm-up phase is done (see section 3.4 for details). If it contains the phrase \texttt{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 \texttt{Generate} process. The network size specifies how many \texttt{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 \texttt{Generate}, a \texttt{Square} and a \texttt{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 \texttt{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 \texttt{TriangleGenerator} process. The network size specifies how many layers of \texttt{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 \texttt{SierpinskiSelector} process. In each of the following layers there are three times as much \texttt{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:

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

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

<table>
<thead>
<tr>
<th>network size</th>
<th>number of processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>123</td>
</tr>
<tr>
<td>6</td>
<td>366</td>
</tr>
<tr>
<td>7</td>
<td>1095</td>
</tr>
<tr>
<td>8</td>
<td>3282</td>
</tr>
<tr>
<td>9</td>
<td>9843</td>
</tr>
<tr>
<td>10</td>
<td>29526</td>
</tr>
</tbody>
</table>

3.3 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, for Java objects is allocated on the heap. This is also true for component objects representing FBP processes. On the other hand, the \texttt{Component} class inherits from \texttt{Thread}. Java threads are implemented using operating system threads on Linux. On invoking the \texttt{start()} methods of the \texttt{Thread} objects, an operating system thread is created and started which executes the \texttt{run()} method. This means, creating a \texttt{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:

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

This happened using a network with 700002 processes. A smaller one with 600002 processes would run. According to Sun this error may occur when the heap is too small, 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:

```java
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.

---

1 These strange numbers occur because there are always two more processes in a serial network than specified on the command line – see section 3.2.5.
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 ⇒ 600000 processes possible
2. 2600 MB heap, connection size = 1, 1 IP per process triple ⇒ 1800000 processes possible
3. default settings, connection size = 10000, 10000 IPs per process triple ⇒ 6000 processes possible
4. 64 kB stack size, connection size = 10000, 10000 IPs per process triple ⇒ 21000 processes possible

As stated before, the main thread is responsible for starting most of the processes. Each Generate process will only start the connected Square process on the first send (see section 2.2.2), 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 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 3.2.5, 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.

3.4 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.

3.4.1 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 (short: JIT) compilation,
- optimization mechanisms in the Java virtual machine (short: 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 (short: 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 $\alpha=0.95$. 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.
3.4.2 Tools

To test the three network topologies, the test program mentioned in section 3.2.5 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](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](http://www.gnuplot.info)).

3.4.3 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:
<table>
<thead>
<tr>
<th>number of processes</th>
<th>run time/ms</th>
<th>size of confidence interval/percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5.42</td>
<td>20.5512701893</td>
</tr>
<tr>
<td>22</td>
<td>10.34</td>
<td>29.0156300695</td>
</tr>
<tr>
<td>32</td>
<td>13.6</td>
<td>6.20474864399</td>
</tr>
<tr>
<td>42</td>
<td>19.2</td>
<td>13.8596459249</td>
</tr>
<tr>
<td>52</td>
<td>24.02</td>
<td>10.9173242847</td>
</tr>
<tr>
<td>62</td>
<td>28.72</td>
<td>13.2612976953</td>
</tr>
<tr>
<td>72</td>
<td>34.3</td>
<td>13.5424403395</td>
</tr>
<tr>
<td>82</td>
<td>35.74</td>
<td>3.90628190267</td>
</tr>
<tr>
<td>92</td>
<td>43.78</td>
<td>10.982907905</td>
</tr>
<tr>
<td>102</td>
<td>46.88</td>
<td>10.4144080463</td>
</tr>
<tr>
<td>202</td>
<td>95.86</td>
<td>4.98112624767</td>
</tr>
<tr>
<td>302</td>
<td>142.98</td>
<td>4.74565682636</td>
</tr>
<tr>
<td>402</td>
<td>198.2</td>
<td>8.79568711548</td>
</tr>
<tr>
<td>502</td>
<td>243.08</td>
<td>5.73481190018</td>
</tr>
<tr>
<td>602</td>
<td>278.24</td>
<td>3.38714283549</td>
</tr>
<tr>
<td>702</td>
<td>333.26</td>
<td>5.61506296307</td>
</tr>
<tr>
<td>802</td>
<td>394.74</td>
<td>5.88969814399</td>
</tr>
<tr>
<td>902</td>
<td>435.08</td>
<td>3.9742907895</td>
</tr>
<tr>
<td>1002</td>
<td>464.06</td>
<td>3.6268833587</td>
</tr>
<tr>
<td>2002</td>
<td>957.72</td>
<td>2.68931730721</td>
</tr>
<tr>
<td>3002</td>
<td>1403.5</td>
<td>2.11321964033</td>
</tr>
<tr>
<td>4002</td>
<td>1914.9</td>
<td>2.11321964033</td>
</tr>
<tr>
<td>5002</td>
<td>2346.21052632</td>
<td></td>
</tr>
<tr>
<td>6002</td>
<td>2889.14285714</td>
<td></td>
</tr>
<tr>
<td>7002</td>
<td>3301.11764706</td>
<td></td>
</tr>
<tr>
<td>8002</td>
<td>3778.97058824</td>
<td></td>
</tr>
<tr>
<td>9002</td>
<td>4276.7</td>
<td></td>
</tr>
<tr>
<td>10002</td>
<td>4729.28571429</td>
<td></td>
</tr>
</tbody>
</table>

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 (3.4.3) 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.

Figure 3.1: serial network, 12 to 1002 processes

If we zoom in, so that only numbers from 12 to 1002 or 12 to 102 are visible (figures
3.4.3 and 3.4.3), 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 3.2: serial network, 12 to 1002 processes

Figure 3.3: serial network, 12 to 102 processes

Testing the same network, but with generating 10000 IPs, yields the following results:
The plot of these values (figure 3.4.3) 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 3.4.3 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 3.4.3), 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 3.4: serial network, 12 to 5002 processes
3.4.4 Parallel Networks

The 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:
<table>
<thead>
<tr>
<th>number of processes</th>
<th>run time/ms</th>
<th>size of confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>9.3</td>
<td>10.6397519147</td>
</tr>
<tr>
<td>60</td>
<td>19.98</td>
<td>47.4620711253</td>
</tr>
<tr>
<td>90</td>
<td>26.7</td>
<td>6.17663500666</td>
</tr>
<tr>
<td>120</td>
<td>35.58</td>
<td>7.68575032386</td>
</tr>
<tr>
<td>150</td>
<td>45.28</td>
<td>9.8712432191</td>
</tr>
<tr>
<td>180</td>
<td>54.74</td>
<td>14.3400441855</td>
</tr>
<tr>
<td>210</td>
<td>61.08</td>
<td>4.95931721377</td>
</tr>
<tr>
<td>240</td>
<td>73.42</td>
<td>10.5976541966</td>
</tr>
<tr>
<td>270</td>
<td>79.82</td>
<td>7.59313096197</td>
</tr>
<tr>
<td>300</td>
<td>87.18</td>
<td>2.9266863959</td>
</tr>
<tr>
<td>600</td>
<td>176.64</td>
<td>4.24446157381</td>
</tr>
<tr>
<td>900</td>
<td>263.98</td>
<td>5.84164402202</td>
</tr>
<tr>
<td>1200</td>
<td>381.42</td>
<td>22.234243256</td>
</tr>
<tr>
<td>1500</td>
<td>455.18</td>
<td>6.74360837741</td>
</tr>
<tr>
<td>1800</td>
<td>542.48</td>
<td>4.14848037104</td>
</tr>
<tr>
<td>2100</td>
<td>628.32</td>
<td>3.89091336399</td>
</tr>
<tr>
<td>2400</td>
<td>719.12</td>
<td>3.99630180021</td>
</tr>
<tr>
<td>2700</td>
<td>803.3</td>
<td>3.1290448444</td>
</tr>
<tr>
<td>3000</td>
<td>896.56</td>
<td>2.47327460943</td>
</tr>
<tr>
<td>6000</td>
<td>1804.1</td>
<td>2.70246711382</td>
</tr>
<tr>
<td>9000</td>
<td>2672.5777778</td>
<td>6.34619360435</td>
</tr>
<tr>
<td>12000</td>
<td>3612.02</td>
<td>6.34619360435</td>
</tr>
<tr>
<td>15000</td>
<td>4494.9</td>
<td></td>
</tr>
<tr>
<td>18000</td>
<td>5333.4666667</td>
<td></td>
</tr>
<tr>
<td>21000</td>
<td>6355.72</td>
<td>5.00715862617</td>
</tr>
<tr>
<td>24000</td>
<td>7279.94</td>
<td>4.24382769937</td>
</tr>
<tr>
<td>27000</td>
<td>7755.6666667</td>
<td></td>
</tr>
<tr>
<td>30000</td>
<td>8935.34285714</td>
<td></td>
</tr>
<tr>
<td>60000</td>
<td>18301.0</td>
<td></td>
</tr>
<tr>
<td>90000</td>
<td>27523.9375</td>
<td></td>
</tr>
<tr>
<td>120000</td>
<td>36887.2142857</td>
<td></td>
</tr>
<tr>
<td>150000</td>
<td>45411.666667</td>
<td></td>
</tr>
<tr>
<td>180000</td>
<td>54594.84375</td>
<td></td>
</tr>
<tr>
<td>210000</td>
<td>63841.733333</td>
<td></td>
</tr>
<tr>
<td>240000</td>
<td>72816.5416667</td>
<td></td>
</tr>
<tr>
<td>270000</td>
<td>82460.7692308</td>
<td></td>
</tr>
<tr>
<td>300000</td>
<td>90782.8</td>
<td></td>
</tr>
<tr>
<td>600000</td>
<td>183073.8</td>
<td></td>
</tr>
</tbody>
</table>

The plots of these values (figures 3.4.4, 3.4.4, 3.4.4, 3.4.4 and 3.4.4) 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.

Testing parallel networks where each process triple processes 10000 IPs yields the following results:
Figure 3.7: parallel network, 30 to 600000 processes

![Parallel Network Diagram](image1)

Figure 3.8: parallel network, 30 to 300000 processes

![Parallel Network Diagram](image2)

<table>
<thead>
<tr>
<th>number of processes</th>
<th>run time/ms</th>
<th>size of confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>3753.66</td>
<td>5.83704732979</td>
</tr>
<tr>
<td>60</td>
<td>5938.0</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>8487.636364</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>11024.63636</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>13703.4</td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>16310.7142857</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>18911.366364</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>21801.6675</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>24856.78</td>
<td>3.97268936762</td>
</tr>
<tr>
<td>300</td>
<td>27522.78</td>
<td>4.02221473391</td>
</tr>
<tr>
<td>600</td>
<td>56766.8333333</td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>86836.2105263</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>116018.071429</td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td>148097.615385</td>
<td></td>
</tr>
<tr>
<td>1800</td>
<td>185735.0</td>
<td></td>
</tr>
<tr>
<td>2100</td>
<td>22081.44</td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td>263919.925926</td>
<td></td>
</tr>
<tr>
<td>2700</td>
<td>298352.75</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>343763.583333</td>
<td></td>
</tr>
</tbody>
</table>
3.4.5 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:

<table>
<thead>
<tr>
<th>number of processes</th>
<th>run time/ms</th>
<th>size of confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.76</td>
<td>68.6891590605</td>
</tr>
<tr>
<td>6</td>
<td>3.54</td>
<td>22.0511347679</td>
</tr>
<tr>
<td>15</td>
<td>7.34</td>
<td>26.807454829</td>
</tr>
<tr>
<td>42</td>
<td>16.66</td>
<td>13.8131740235</td>
</tr>
<tr>
<td>123</td>
<td>47.24</td>
<td>13.3973457317</td>
</tr>
<tr>
<td>366</td>
<td>174.36</td>
<td>6.64456873188</td>
</tr>
<tr>
<td>1095</td>
<td>930.14</td>
<td>5.34448891962</td>
</tr>
<tr>
<td>3282</td>
<td>6354.06</td>
<td>6.38294846484</td>
</tr>
</tbody>
</table>

Looking at the plots (figures 3.9 and 3.10) reveals a very similar scaling behavior like in the respective serial test case. Also in this case JavaFBP scales a little worse than linearly.
3.5 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.

3.5.1 Increasing Memory, Distribution

Section 3.3 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 1.3.1, 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 1.4.4 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.

### 3.5.2 Cooperative M:N Scheduling

In section 3.4 it was shown that JavaFBP does not scale well when a network consists of many active processes 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. 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 preemptively
scheduled by the operating system. 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 Ierusalimschy [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 facil-
ity 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 sub-
mitted 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 run on 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 continuation, 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 Srini-vasan [2006], section 2). One-shot continuations are less computationally expensive but also less powerful. But they suffice to implement coroutines [de Moura and Ierusalim-schy [2004], section 4.2]. Because they are faster than multi-shot continuations [Brugge-man 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.
A Source Code

A.1 JavaStats

A.1.1 javastats.conf

```plaintext
[general]
benchmark suites: serial parallel triangle
virtual machines: sun
output file: results

[trace]
machine: localhost
location: /tmp
prefix: javastats_trace

[performance]
class: TimePerformance

[stats]
minimum vm invocations: 10
maximum vm invocations: 50
minimum benchmark iterations: 1
maximum benchmark iterations: 1
confidence level: 0.95
stop criterium: percentage
stop threshold: 2.0

[sun]
binary location: /usr/bin
binary name: java

[serial]
location: /home/master/Masterarbeit
input sizes: 6000 7000 8000 9000 10000 20000 30000 40000 50000 60000 70000 80000 100000 200000 300000 400000 500000
startup command: -XX:+UseParallelGC -server -jar Test.jar sequential %(benchmark)s %(input)s 10
steady command: -XX:+UseParallelGC -server -jar Test.jar sequential %(benchmark)s %(input)s 10
benchmarks: 10000
ulimit threshold:

[parallel]
location: /home/master/Masterarbeit
input sizes: 10 20 30 40 50 60 70 80 90 100 200 300 400 500 600 700 800 900 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 20000 30000 40000 50000 60000 70000 80000 100000 200000 300000 400000 500000
startup command: -XX:+UseParallelGC -server -jar Test.jar parallel %(benchmark)s %(input)s 10
steady command: -XX:+UseParallelGC -server -jar Test.jar parallel %(benchmark)s %(input)s 10
benchmarks: 1 10000
ulimit threshold:

[triangle]
```

A.1.2 performance.py

```python
#!/usr/bin/env python

# JavaStats is a toolkit designed to get a
# statistically rigorous performance evaluation for a
# given (Java) application
#
# Copyright (C) 2007 Andy Georges
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
# 02110-1301, USA.
#
# This file implements several useful example snippets that can be used to
# return a desired performance number to the main JavaStats monitor. Basically,
# you need to implement two methods given in the Performance base class:
# acquire_command and get_performance_data. The former will adjust the java
# run command and set it up to gather the desired data. The latter will examine
# the resulting trace file and fetch the desired data.
#
import re

class Performance:
    """
    Performance is the base class for determining the performance
    according to some criterion (e.g., time, executed instruction,
    etc.) of a Java execution.
    """
    # regular expression that check if an error occured during a run
    # as well as the regex for the runtime in startup mode
    re_exception = re.compile("[Ee]xception")
    re_error = re.compile("Error")
    re_deadlock = re.compile("deadlock")
    re_terminated = re.compile("terminated_by")
    re_non_zero = re.compile("non_zero_status")
```

re_stack = re.compile("Stack")
re_failed = re.compile("FAILED")
re_not_valid = re.compile("NOT_VALID")

def acquire_command(self, command):
    pass

def get_performance_data(self, trace_filename):
    pass

def sweep_line_for_error(self, line):
    """The sweep_line_for_error function will try to find selected regular expressions in the line that indicate an error or unexpected situation occurred during the execution. If that is the case, the experiment should be discarded."
    if self.re_exception.search(line):
        return True
    if self.re_error.search(line):
        return True
    if self.re_deadlock.search(line):
        return True
    if self.re_terminated.search(line):
        return True
    if self.re_non_zero.search(line):
        return True
    if self.re_stack.search(line):
        return True
    if self.re_failed.search(line):
        return True
    if self.re_not_valid.search(line):
        return True
    return False

class TimePerformance(Performance):
    re_time = re.compile (^[0-9]+[0-9]+[0-9]+[0-9]+$)

def acquire_command(self, java_command):
    return java_command

def get_performance_data(self, trace_filename):
    f = file(trace_filename, 'r')
    for line in f.readlines():
        if self.sweep_line_for_error(line):
            return None
        if self.re_time.search(line):
            return float(line.strip().split("\n")[-1])
Bibliography


