The ToolBus:
Inter Tool Communication

Arnold Lankamp

08-14-2006

One Year Master Program Software Engineering

Thesis Supervisor: Paul Klint

Internship Supervisor: Paul Klint

Company or Institute: CWI

Availability: Public domain

University of Amsterdam

Content

Summary

This project is about improving the scalability of the ToolBus. The ToolBus is a coordination architecture that enables components (also referred to as tools), to communicate in a platform and programming language independent way.

One of the main problems with the current implementation of this architecture is that all the communication is routed through the ToolBus. Normally this is not a problem, but when large objects are being transmitted it will lead to problems, since these objects need to be stored in memory. Due to the 2 GigaByte memory limit of 32-bit applications, this is a scalability issue. Additionally, every object needs to be sent twice; once to the ToolBus and once to the tool that needs it. This could be handled more efficiently.

To solve these two problems we propose to enable tools to intercommunicate with as goal to decrease transportation times and conserve memory at the ToolBus. In the new situation tools will only send a 'reference' to the ToolBus, stating where to find the associated value and what the value 'looks like'. Whenever the ToolBus sends a reference to a tool, the receiving tool will retrieve the value and acknowledge its reception to the ToolBus. Since we are now only sending references to the ToolBus, its memory usage will decrease considerably. Furthermore, objects can now be directly exchanged between tools, resulting in an overall decrease of network traffic and offloading the connection of the ToolBus. The ToolBus will only be in charge of the coordination between tools; this is what it was designed for. Because we are introducing references, we will need to take garbage collection into account. Reference counts will be added to the 'variables' that are present at the ToolBus. Whenever a variable becomes unreachable, the tool in possession of its value will be notified that it can be reclaimed.

During this project a prototype was implemented to prove the viability of the solution for implementation in the ToolBus and for executing benchmarks on. The results of the benchmarks are impressive. The memory usage of the ToolBus did not exceed 5 MegaBytes in any of the tests. Transportation times were also cut considerably.

Besides this additional improvements were also implemented. Most notable are:

- Improved concurrency. We are now able to utilize the available resources more efficiently. This goes for both processing power as network bandwidth. The increase in performance will be most noticeable on multi-core / processor systems.
- More advanced serialization support. The prototype uses custom made serializable objects, instead of ATerms. The current Java implementation of the ATerm library lacked performance and did not offer the functionality required for usage in a high performance coordination architecture. These serializable objects are several times as fast as ATerms and a lot more memory efficient.

During this project we proved that the proposed solution is viable for usage in the Next Generation ToolBus. Its benefits are clear and will increase the scalability of the ToolBus considerably. Changes to the ATerm library are however required to make the implementation of the solution, which is proposed in this thesis, possible.

Preface

Introduction

In this thesis we will take a look at the project I did at the CWI (Centrum voor Wiskunde en Informatica / Center for Mathematics and Computer science). This project was about finding a solution for scalability problems associated with transferring large objects between programs. We will elaborate on this problem further in the next chapter. In the second chapter we will place this problem in context and discuss the relation it has with existing techniques. After which we will take a look at how we decided to handle this project and how this plan was executed. This will result in an explanation about the solution that was developed and its associated benefits in relation to the current implementation; this will be shown by means of a series of benchmarks. To conclude this thesis an evaluation will be given at the end, highlighting the most important things discussed in previous chapters.

Word from the author

It might be interesting to know why I chose to do this project. First of all, I had already worked at a small and a large company during previous internships. So I decided to try a different setting once again, to better orient myself. An additional advantage is that I was ensured of proper guidance and would not be interrupted by anything, which is very much the case in small companies. The project itself appealed to me, because it seemed challenging. I did not have an idea about how to solve it exactly before I started. Which was refreshing for a change. Besides that I wanted to do a complicated technical project. This project fitted that description perfectly.

Intended audience

This thesis is aimed at everyone who is interrested in reading about how to optimize the performance of a bus architecture, students of the Software Engineering course on the university of Amsterdam and the people that have contributed to the developement of the ToolBus.

Acknowledgement

I would like to thank everyone that supported me and contributed to the project's success. (Now I can be sure I did not forget anyone).

Problem description

At the CWI a coordination architecture, called the ToolBus, was developed. This ToolBus enables applications, also referred to as tools, to communicate without requiring them to know each other's interface. There is however a problem with this architecture; sending large objects between tools is highly inefficient, as discussed in [3]. There are a several of reasons for this:
First the object needs to be transmitted to the ToolBus, which stores it in memory, potentially causing resource problems. Furthermore, this seriously limits the scalability of the system. Storing one object of 1GB in memory should not be a problem (assuming we have full access to the systems resources). But if another similarly large object arrives, we have got a problem since the maximum heap size of a 32-bit application is around 1.6 GigaBytes (2 GigaBytes minus thread stacks). Another problem is related to performance, the object needs to be sent two times. Once to the ToolBus and from the ToolBus to the tool that needs it. With small objects this does not pose a problem, but sending reasonably large objects could be handled more efficiently.

The proposed solution for this problem is to enable tools to intercommunicate (*figure 1*), with the sole purpose of exchanging large objects. Effectively cutting the time to transport objects in half and decreasing the strain on the ToolBus and its resources (memory as well as bandwidth). If data can be exchanged between tools directly, it will reduce the amount of data that is send overall; we will only need to send it once instead of twice (under the condition that there is one sending and one receiving tool). Additionally, if the values of the objects no longer need to be send to the ToolBus, it will reduce the memory usage of the ToolBus considerably, since the ToolBus no longer needs to store the value of the objects in memory.

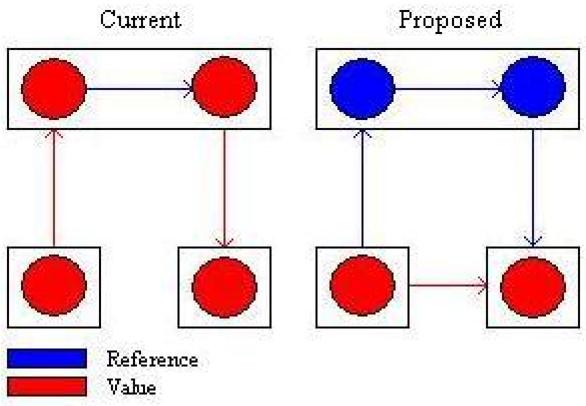

Figure 1: The current and the proposed situation.

There are, however, some complications. First of all, the garbage collection of objects. Since tools need to maintain objects that can potentially be requested by other tools they will need to be informed when these objects are no longer required. The question is how to check which objects are still reachable or not? This information should be maintained by the ToolBus, since it is providing the coordination between tools. Although it could happen that an object can become unreachable at the ToolBus level, but has not been (completely) received by the tool that requested it. In this case deleting the 'unreachable' object will cause a failure. Another complication is that a tool can also crash, become disconnected or unreachable for any other reason; further complicating the solution.

To summarize the above, the memory and bandwidth usage of the ToolBus must be reduced, with as ultimate goal: achieving an improvement in scalability.

Background and context

The ToolBus

Because this project is related to the ToolBus [1][2] it is important that we give a global sketch what it is and what it was designed for.

The ToolBus is a component inter connection architecture. It handles the coordination between different applications, also referred to as tools. This architecture formalizes the interaction between tools by means of ToolBus scripts. These ToolBus scripts contain 'process logic' that describes the actions that need to be performed when a certain event is triggered. The main goal of the ToolBus is to achieve a language independent platform, which enables tools to co-operate without exposing their interface.

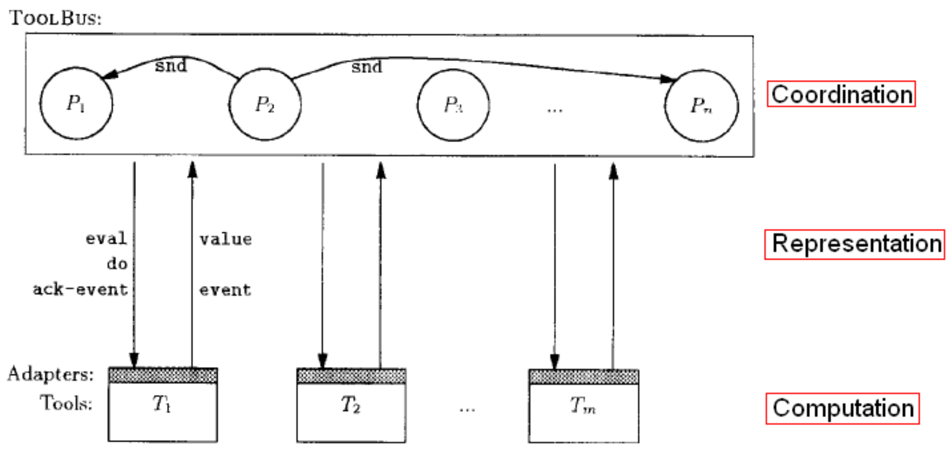

Figure 2: The ToolBus architecture.

Because tools that are written in different languages have to be able to communicate through this bus architecture, ATerms where used as data type. ATerms are designed to exchange data structures in distributed applications in a platform and language independent way [4].

Garbage collection

Performing distributed garbage collection is going to be one of the main issues in this project. Here we will take a look at the main garbage collection techniques [5], their advantages and disadvantages.

Reference counting

Reference counting algorithms come in different variations but the underlying idea is the same. Every object maintains a counter, indicating how many other objects have a reference to it. When that counter reaches zero the object is no longer reachable and can be destroyed.

The main advantage of garbage collection by means of reference counting algorithms is that unreferenced objects will be reclaimed directly. Additionally this type of garbage collection can be done concurrently with the execution of the application.

There is however a major problem, because of the nature of reference counts, reference cycles cannot be reclaimed. Even if the objects that form the cycle have become unreachable from the 'root' of the application, their reference counts will never reach zero since they are referencing each other. Either forbidding reference cycles or running a periodic trace to collect them can prevent this. Another downside is that the counter that they need to hold will bloat objects slightly. The impact this has on memory usage is dependent on the number and the size of the objects. Furthermore the constant increasing and decreasing of the counters also brings a computational overhead along.

There are different variations on the reference counting algorithm. Examples of this are:

- Weighted reference counting [6].
- Generational reference counting [7].
- Indirect reference counting [8].

All of these algorithms strive to improve the performance of reference counting algorithms in a distributed environment, mainly by reducing the necessary communication between nodes. The general idea is to have a local reference count for remotely referenced objects and only send a notification when the object is no longer referenced internally. This is opposed to sending a message every time the reference count is incremented or decremented. Limiting control messages to two per node (obtain and release). This makes this kind of algorithms particularly attractive to use for distributed garbage collection.

Tracing collectors

Tracing algorithms are most common nowadays. Modern languages like Java and C# use them to perform garbage collection. As the name indicates, tracing collectors 'trace' through the hierarchy of objects to determine reachability. First of all they identify the 'root set'. Usually the root set consists of the objects that are referenced by a pointer on the thread stack(s) or by any global or statically accessible variable. From this root set the collector will walk through the object hierarchy marking or copying all reachable objects. After the collector has completed the trace, all the objects that are un-marked / not copied can be reclaimed.

Compared to the reference counting algorithm there are several advantages and disadvantages. The up side is that tracing collectors are complete, meaning that they will always properly detect unreachable objects and thus can reclaim objects containing circular references. The main disadvantage of tracing collectors is that, because they need to trace through the entire object hierarchy and they need to suspend the execution of the application to ensure correctness, performance can become a problem; especially when dealing with applications that use a lot of memory. Besides this, objects will not be reclaimed as soon as they become unreachable since the collection is usually triggered by an event, for example when available memory runs low. This causes unreachable objects to persist in memory for an undetermined amount of time. Note that due to the amount of required communication to determine reachability, the inability to run concurrently with the execution of an application and the lack of promptness, tracing collectors are not suitable for usage in a distributed environment.

However there are different strategies tracing collectors can use to improve their performance. These include:

- Moving versus non-moving collectors.
- Dividing the memory into 'generations'.
- Stop-the-world (including incremental) versus concurrent collection.

Moving versus non-moving
There are two different strategies regarding memory allocation. Moving and non-moving. Moving collectors copy all the reachable objects from one part of the memory to another. This seems inefficient at first, but it does have its advantages. Namely after moving all the reachable objects to a new part of memory everything left in the old part can be reclaimed since it is no longer reachable; this simplifies the reclamation process. Additionally the heap will not become fragmented since all the objects are allocated contiguously. This also greatly simplifies the allocation of new objects; it can just be stored after the previously allocated object, this way the system does not need to maintain a list of free memory blocks. Because of the order in which the objects are copied, related objects will be close together in memory, possibly speeding up access times. This strategy does however have one problem, you will need about twice as must memory for your application, since it needs to copy all the objects from one part of the memory to another. Non-moving collectors, as you might expect, do the exact opposite. They leave all the objects where they are, possibly causing heap fragmentation over time. An example of an algorithm that uses the non-moving strategy is mark-sweep. Although non-moving algorithms can be augmented with peridoc heap compaction, offering a bit of both.

Generations
Another strategy to improve performance is generational garbage collection. Here the heap is divided into different regions, for example a part containing newly allocated objects and a part that contains objects that have been reachable for a longer period of time. The idea is based on the fact that newly created objects will most likely become unreachable quickly. After surviving a number of garbage collection cycles these 'new' objects will be promoted to the next generation. This way the number of objects that needs to be checked on reachability during a collection can be minimized, thus improving performance. A negative influence of this is that it will take longer before unreachable 'long lived objects' are detected.

Pause times
Another main disadvantage that needs to be adressed is the fact that the garbage collector needs to stop the execution of an application if it wants to perform a collection. If it would be possible to allocate new objects and alter pointers during a collection cycle, correctness could not be guaranteed. Especially in (soft) real-time systems it is unacceptable that the execution of an application is paused for an undetermined amount of time. Incremental collectors try to solve this problem, minimizing pause times in exchange for more overhead. They divide the collection into parts, executing it bit by bit. If objects are allocated at a higher rate than they can be reclaimed they will however fall back on the stop-the-world algorithm, preventing out of memory exceptions. Concurrent collectors try something similar and are mainly used on multi-processor systems. They only stop the application when necessary and try to execute as many garbage collection operations as possible concurrently with the execution of the application. They will however introduce some overhead, because they are more complex than the basic algorithm.

Hybrid algorithms

Besides reference counting and tracing collectors there are also certain hybrid algoritms which combine the two. These are mostly experimental and aim to offer the best of both worlds. Most notable is the 'garbage collecting the world' algorithm [9][10], which tries to improve distributed garbage collection based on reference counting by augmenting it with a tracing collector.

Java

The next generation ToolBus is being implemented in Java. The main advantage of Java is that it is platform independent, thus it will increase the applicability of the ToolBus considerably. Besides this, it currently offers some interesting features, which we will take a look at below.

NIO

The communication in the next generation ToolBus implementation is based on NIO (which stands for New I/O) [11]. NIO provides non-blocking, buffered I/O with the ability to multiplex. This enables the construction of highly scalable network applications, since we do not need a separate thread to listen at every socket, but only one to monitor all of them. The down side is that it will make your application more complicated and harder to understand.

References

Java has another interesting feature, related to garbage collection that we might make use of in the implementation of the prototype. Since JDK 1.4, Java supports soft- and weak-references [12]. These references have a different behavior compared to 'normal' ones; the object they point to can be reclaimed at the garbage collectors discretion, provided that the object has become unreachable by traditional means. This can be very useful for implementing (memory sensitive) caches for example. Currently there are four different kinds of references:

- Strong: A regular pointer.
- Soft: Will be reclaimed at the garbage collector's discretion / when memory runs low.
- Weak: Will only remain reachable as long as there are strong references to the object.
- Phantom: Has become unreachable and is awaiting finalization.

This feature might be useful to further improve performance by implementing a constant pool that caches frequently sent objects.

Research plan

Initial idea

As discussed the proposed solution to the given problem is to enable tools to inter communicate with the purpose to exchange large objects directly instead of through the ToolBus.

The idea is to stick to the pure value based transmission of objects between Tools, because this is the case in the current ToolBus implementation. Introducing objects that are reachable by reference would (unnecessarily) increase complexity. On the level of the ToolBus we will, however, introduce references. These references will contain information about where to find the associated value. When the ToolBus wants to send the object to a Tool, it will transmit the reference to that Tool, which will in turn retrieve the value from the tool that has the associated value in its possession. The references in the ToolBus will be given a reference count. When the reference count reaches zero, the tool that is in possession of the original value will be notified, so it can take appropriate measures to make it subject to collection.

Sounds easy enough, but there are still some problems that need to be addressed.

First of all, it suffers from the same problem as all other reference counting solutions do; when a tool crashes, gets disconnected or becomes unreachable for any other reason, the reference count will never be decremented, preventing the value object to become subject to collection. Adding leases to the references, as an intermediate solution, could solve this. When the lease expires, we assume that the associated object is no longer required, since it has not been accessed for a long period of time. All the outstanding references will be invalidated so the object can be collected and its associated resources released. This way we can ensure that there will be no memory leaks due to reference counts that are not properly decremented; No matter what the cause. We should, however, be cautious with leases, because they might also cause reachable objects to be destroyed if they have not been accessed for a long period of time.

An even bigger problem is the fact that we do not know when an object has arrived at the intended destination. This can cause a problem, because it is possible that an object becomes unreachable at the ToolBus level, while (or even before) its value has been transferred between the tools. This may cause the value to be deleted even though it is still needed. This matter is further complicated, because a tool could request the value multiple times. As off yet, there is no suitable solution for this problem.

Besides the problems related to garbage collection, sending a value between tools through tool inter communication introduces some overhead and will not be the fastest way to transmit an object in all situations. This is especially the case with small objects. Therefore it would be best if we let the way of transmission depend on the size of the object that needs to be sent. When it is small enough we will incorporate the value in the message and in any other case a reference will be passed to the ToolBus. The optimal threshold for the object size will be determined by a series of benchmarks. It will however be adjustable at the users discretion, because the performance will vary depending on the available bandwidth.

When there is time

Apart from the main problem that needs to be solved, there are some other (somewhat related) things that might be interesting to take a look at.

Streams might be interesting to experiment with, since it would reduce the amount of memory that is required to send an object. Especially for BLOBs[1], which are currently causing problems. If we use streams to send objects across it will be done byte-by-byte, which only requires a buffer that contains the received bytes that have not been sent yet. This is more memory efficient than having to contain the entire object in memory before being able to send it to its intended destination.

Security is also an issue at the moment, because there is none. Java supports SSL sockets, so why not use them?

Implementation

The solution will be implemented in a separate system (a.k.a. Prototype). This system will emulate the ToolBus behavior as closely as possible, to ensure the solution is realistic and can be implemented in the 'real' ToolBus. The decision to implement it in a separate system was made, because there is only a limited amount of time available. It will reduce dependencies and thus complications, making the project go more smoothly. No time is 'wasted' on understanding an existing implementation. We can also reasonably expect that the existing system will have some design incompatibilities, which would cause a serious delay, because parts of is will need to be redesigned and re-implemented. An additional benefit is that it will be easier to compare performance, since the benchmarks will be run on the same system. The only thing that is different is the way of transmission. Removing any external factors that could have an influence on the results.

Expected results

This project will have the following deliverables:

The architecture and the design

It would be handy to have some kind of documentation about the solution. Which decisions were made, how they were implemented and, if applicable, what limitations there are. All these points will be discussed in this thesis.

A proof of concept

A prototype will be made in which the solution is implemented, to prove that it is viable for usage in the 'real' ToolBus. This prototype will also be used for benchmarking purposes and could serve as a base for testing future optimizations related to the solution.

A number of benchmarks

The results of the benchmarks between the current and the 'new and improved' way of transportation will be presented. These benchmarks will at least consist of:

- A memory usage comparison, including the ToolBus and the connected Tools.
- A throughput comparison, showing the relative gain in performance.
- A 'turning point' analysis, discussing how to determine what the optimal threshold is for determining how to send data (by value or through tool inter communication).

[1] Binary Large OBject

Required expertise

To successfully complete this project I will need to gain insight in the following subjects:

The ToolBus

First of all I will need to get familiar with the technical details of the ToolBus and specifically about how it is implemented. This knowledge is required because you cannot develop a solution for something you do not fully understand. Apart from that, I will need to implement a prototype that resembles the current ToolBus implementation.

Inter process communication

Secondly, knowledge about how process intercommunication works and what problems are associated with it, is important. I need to know what the technical possibilities and limitations are.

Distributed garbage collection

The same goes for distributed garbage collection; if you do not know the limitations that are associated with the different algorithms, you will never be able to get a correctly working solution that adheres to the requirements.

Strong knowledge of Java programming

The prototype will be implemented in Java, like the next version of the ToolBus. Because the ToolBus is not just some everyday system and the solution will be relatively complex as well, strong knowledge about Java programming is mandatory.

Risks

Every project is unique and you never know what to expect. Risks are associated with these uncertainties. The trick is to identify and manage those risks, so you do not run in to any unpleasant surprises. I identified the following risks:

At the moment of writing this plan I have only limited insight in the current implementation of the ToolBus. The ToolBus is what this project is about, so this is quite a fundamental problem. I will however be given the source code of the current Java implementation of the ToolBus; it should not take long to figure out how it works.

Another limitation is related to time constraints. There are only twelve weeks available to complete the project, which is a fairly short amount of time. I am trying to make things go smoother by limiting the amount of dependencies; for example by deciding to implement the solution in a separate system, as discussed above. Dependencies are directly related to the amount of problems you will run in to and have no influence on.

Lastly, another risk is: the inability to find a perfect solution within the given time. Good is not good enough for me, I would like to do it the best possible way imaginable. This will be though within the limited time available. For example, the leases that where discussed previously are a weakness; I am not too happy about those for obvious reasons, but I have not been able to find a better solution as of yet.

Plan execution

Introduction

In this chapter we will take a look at how the project developed. Which problems we ran into and how these were solved. We will also take a look at the solution we ultimately ended up with and discuss its implementation in detail. Finally, some propositions about additional improvements will be made, both related and unrelated, to the central research question.

The problem

Before this project started I had more or less worked out a solution, as stated previously. The problem was however that I did not have any insights into the current implementation. This was the first thing that I needed to work on. What I discovered was that the solution I had in mind was, unfortunately, not going to work without some changes. It already had some holes and weaknesses, so I had to think of something better.

The solution

Surprisingly it did not take long to figure out how to solve all of these problems. Basically the original idea more or less formed the base of the solution. Here we will take a look at:

- The modifications that where made in relation to the original idea.
- The problems that still needed to be addressed.
- The solutions for the given problems.

Variables

First of all I wanted to achieve complete transparency; semantically it should not matter if something is a value or a reference, the behavior should be exactly the same. Both for the ToolBus as for the tools. To achieve this, the notion of variables was introduced. These variables can be exchanged between tools and the ToolBus; there purpose is to provide access to there associated (remote) value. When a variable is present on the ToolBus level it can be directly linked to an 'ordinary variable' used inside a ToolBus script. They contain either the 'value' of there associated term or a reference containing information about where to find that value. In case the content of the variable is requested one of two things can happen:

- The value is present and is returned directly.
- In case the variable contains a reference, the value is retrieved from the remote location, cached internally, and then returned.

This way, the caller does not need to know whether the variable contains a reference to a remote value or the value itself. An additional advantage that the caching of values brings along is that we can ensure that it will be retrieved only once. This provides a serious performance gain when a tool requests the value multiple times, although this is not expected to happen often. The main advantage is that it simplifies things, because it ensures consistent behavior, independent of the number of times the value is requested.

However, the process logic present in ToolBus scripts relies on the types and composition of terms. Since the value of the terms is not always present in the new situation, we will need to represent this information in a different way and store this in the variable, regardless of whether the variable contains a value or not. These 'signatures' should be provided by terms themselves; including a facility to enable the matching of signatures, as required by the process logic in the ToolBus scripts. The current ATerm implementation already provides some matching facilities, but these require the values of the terms to be present and thus will need to be extended. These matching facilities are required, because currently the statements in the ToolBus processes rely on the matching of terms.

We will explain how this matching works by means of an example. Below a very simple ToolBus script is shown. The is a tool called control, which can only do one thing and that is save a file when a 'save' event is triggered. Besides this process another one is running that is able to retrieve a text when the `get-text` 'function' is called.

```
tool control is {command = "wish-adapter -script control.tcl"}

process Control is
let
   Control : control,
   Text : str,
   File : str

in
   execute( control, Control? ) .
   (
      rec-event( Control, save( File?) ) .
      snd-msg( get-text ) .
      rec-msg( get-text( Text? )) .
      snd-ack-event( Control, save( File )) .
      snd-do( Control, writeFile( File, Text ))
   +
      rec-disconnect( Control ) . shutdown("")
   ) * rec-note(killed) . shutdown("")
endlet
```

We will walk through what this script does here. To initiate the save event the associated 'control tool' needs to send a term to this process that could for example look like this: save ("/home/myname/filename.txt"). This term will match with the following statement in the script: `rec-event( Control, save( File? ) )`. The stated term is an event, generated by the associated control tool and it's signature matches the save function, which accepts a string. The question mark behind `File` indicates that the instance variable will be set to the received value if an event arrives that matches this signature. After receiving this event the `get-text` function of another process is called, which returns a term that could look like this: get-text("This is a text"). This matches the statement: `rec-msg( get-text( Text? ))`, since it is a message that is received from another process and the signature matches. After acknowledging the completion of the event, a request is sent to the associated control tool to write the text to a file. The term contained in this request might look like this: writeFile("/home/myname/filename.txt" , "This is a text").

This is basically how matching works. Since matching only relies on the types and the composition of the terms, signatures that, for example, look like 'get-text(<str>)' are sufficient. Keep in mind that this is just an example, the exact representation is up to the developers of the ATerm library.

Reachability

The biggest problem that needs to be solved is that there is no way to know for sure when a value can be 'released', as discussed in the previous chapter. A tool could be in possession of a reference to an object, but have not received the value yet. In case this object becomes unreachable at the ToolBus level, the tool that is in possession of the value will be notified enabling the value to be reclaimed. If the tool that is in possession of the reference tries to retrieve the object after it has been deleted, an error would occur, because it is not longer present. This is, of course, unacceptable. The solution for this problem is relatively easy. When a process instance requests a variable to be sent to a tool, it will block the execution of the next statement in that specific process instance until the tool responds with an acknowledgment, confirming that is has completely received the value. Semantically this is conformant to the current ToolBus implementation. This way we can be positive that when a variable becomes unavailable on the ToolBus level, it will no longer be requested by any tool, since it will not be possible to gain possession of a reference to an object from that point on. There is, however, one precondition that needs to be fulfilled, namely that a value will only be retrieved once per variable. Fortunately we have that insurance, because retrieved values are cached, as discussed previously.

Now that we have this insurance, we have to think of a way to identify unreachable objects on the ToolBus level. The first idea is to rely on the garbage collector that is running on the ToolBus, to determine whether an object has become unreachable. If the finalizer on a variable present at the ToolBus level is invoked steps will be taken to inform the tool that is in possession of the value, so it knows that it is no longer required. Although this works it is not prompt enough. There is a variable delay between the time that the object becomes unreachable and the time it is discovered that the object is unreachable. The time of these delays is dependent on the speed that new objects are allocated and the type of garbage collection algorithm that is used. It is possible to send a request to the garbage collector to do a collection. If this is done at regular intervals we should get some insurance about the maximum time it takes to discover an unreachable variable. The problem is however that the garbage collector is free to ignore these requests; manual invocation of the garbage collector could even be disabled completely. Additionally, running a full garbage collection is quite a costly operation. Because we cannot ensure timely collection of unreachable variables, which could result in memory problems for the tools that are in possession of large values, this solution is not good enough to be used. What is done instead is adding reference counts to the variables that are present at the ToolBus level. When a variable is made available to a process instance as an 'instance variable' the reference count is increased, when it is removed it is decreased. The same goes for variables that are transferred to Tools, when it is send the count is increased and as soon as the acknowledgement from the receiving tool arrives the count is decreased. This way, even when an 'instance variable' on a process instance is concurrently modified while the value of the variable is being transferred between tools, it will remain reachable until we are notified that the receiving tool has completely received the value. Because of the nature of reference counts, we are able to tell when a variable becomes unreachable without any delays. An additional advantage of this solution is the fact that, semantically, it behaves in exactly the same way as the current ToolBus implementation. The statements in a process instance are executed sequentially and the next statement in a process instance will not be executed until the value that is being transferred has completely arrived.

The protocol

To facilitate the tool intercommunication and the transferring of variables the current ToolBus protocol needed to be extended. Here we will describe when and in what order certain messages will be sent, what exactly will be transferred. First we will discuss what kinds of messages there are and what their purpose is. Afterwards we will walk through the different scenarios that require communication. Keep in mind that the protocol described in this chapter is the one that is implemented in the prototype and does not completely correspond with the protocol as it is currently implemented in the Toolbus. It will however give an indication about the changes that need to be made to the ToolBus protocol to enable tool intercommunication.

There are six kinds of messages:

Operation	REG	
Summary	This is a message that the ToolBus sends to a tool that has just connected. It enables a tool to be uniquely identified.	
Content	Field	Size
	Tool identification	8 bytes (long)

Operation	PUT	
Summary	This is a message that contains information about where to find the value of a variable. Optionally the value may be present in this message as well.	
Content	Field	Size
	Transaction identification	8 bytes (long)
	Variable identification	8 bytes (long)
	Source tool identification	8 bytes (long)
	Source tool host IP	4 bytes (IPv4)
	Source tool port number	4 bytes (integer)
	Term signature	4 bytes for size specification, 2^32 limit for the signature (integer + 2^32 bytes)
	Term value (optional)	4 bytes for size specification, 2^32 limit for the data (integer + 2^32 bytes)

Operation	GET	
Summary	This is a message that can be sent as a request for retrieving the value of a variable.	
Content	Field	Size
	Variable identification	8 bytes (long)
	Source tool identification	8 bytes (long)
	Target tool host IP	4 bytes (IPv4)
	Target tool port number	4 bytes (integer)

Operation	ACK	
Summary	This message is sent to acknowledge the reception of a variable, including its value.	
Content	Field	Size
	Transaction identification	8 bytes (long)

Operation	FIN	
Summary	This message is sent to a tool to notify it that all the references to the value it was holding have become unreachable.	
Content	Field	Size
	Variable identification	8 bytes (long)

Operation	END
Summary	This message is sent to notify the 'communication partner', that the connection should be closed. This 'communication partner' can be either the ToolBus or a Tool.
Content	No content, this is an empty message.

All of these operations will be contained in a package that is conformant to the following format:

Content	Size	Description
Length	4 bytes (integer)	Specifies the length of the data that is coming
Operation identifier	3 bytes (3 characters)	Specifies the operation
Data	2^32 bytes limit	Contains the serialized representation of the operation

Communication scenarios

Here we will take a look at all the different scenarios related to communication.

Connect

After a connection with a tool is established, it will need to be uniquely identifiable. After generating a unique identifier at the ToolBus level a REG message will be sent to the tool that just connected, so the tool can incorporate it in every message it sends. This is necessary, because we want to be able to tell where a message originated. The combination of the tool identifier and variable identifier is always unique.

Sending a value

Sending a value will go relatively similar to the way it is done currently. A completely constructed PUT message is created, meaning that the data is present as well. This PUT message will be serialized and sent to a tool or the ToolBus. This way of communication will still be possible and will mainly be used for sending small values, since the overhead associated with inter tool communication will yield a bigger penalty than just sending the value directly.

Sending a value through inter tool communication

Sending a value through inter tool communication can be done in three separate ways, depending on connectivity limitations due to firewalls and routers:

- The requesting tool can retrieve the value directly (active[2] -> active or passive[3] -> active)
- The requesting tool can issue the request throught the ToolBus (active <- passive).
- The requesting tool can access it through the ToolBus (passive - passive).

The sequence diagrams are shown below in the order stated above.

The underlying idea is the same in all cases. A reference to a variable is sent to a tool by means of a PUT message, which will retrieve the data upon reception. To do so it will send a GET request to the tool that is in possesion of the value of this variable. All the information required for issuing this request is present in the put message. The response to the get request will be another PUT message, containing the value of the variable. After the reception of the value a ACK message will be sent to the ToolBus, confirming the completion of the transmission.

[2] Active: means that the 'node' is reachable and a connection can be established with this node.

[3] Passive: means that the 'node' is unreachable (because it is behind a firewall or router for example) and no connection can be established with this node. Passive nodes are only able to establish a connection with active nodes. Passive to passive connections are not possible.

Active -> active or passive -> active.
When the tool that is in possesion of the value we wish to retrieve is reachable we can retrieve the value directly.

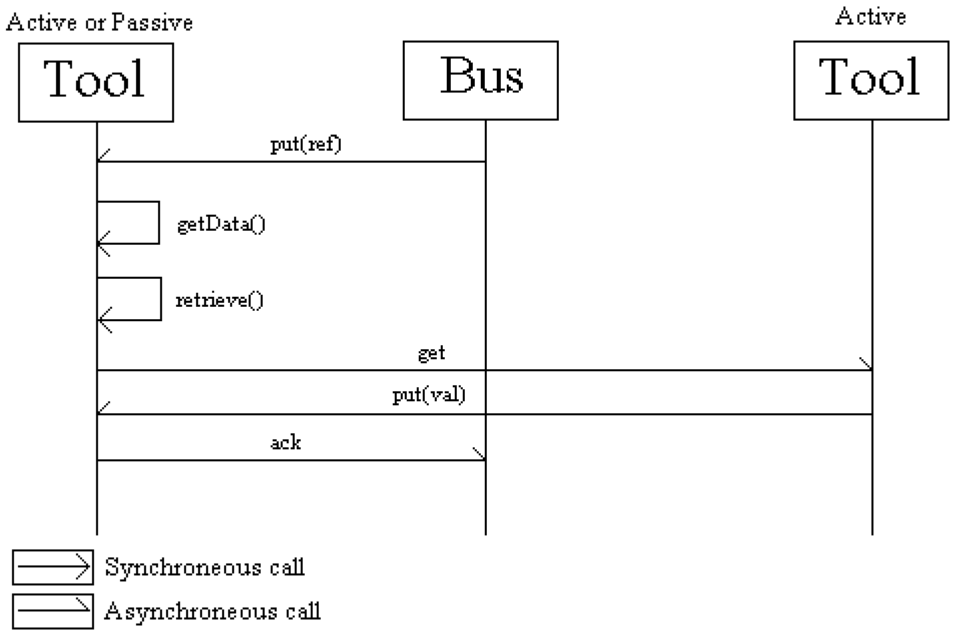

Figure 3: Sequence, active -> active or passive -> active.

Active <- passive.
If the tool that is in possesion of the value we wish to retrieve is not reachable, we need to issue the request to the ToolBus, which will in turn relay it to the concerning Tool. The tool will then try to establish a connection with the requesting tool, which is in this case reachable.

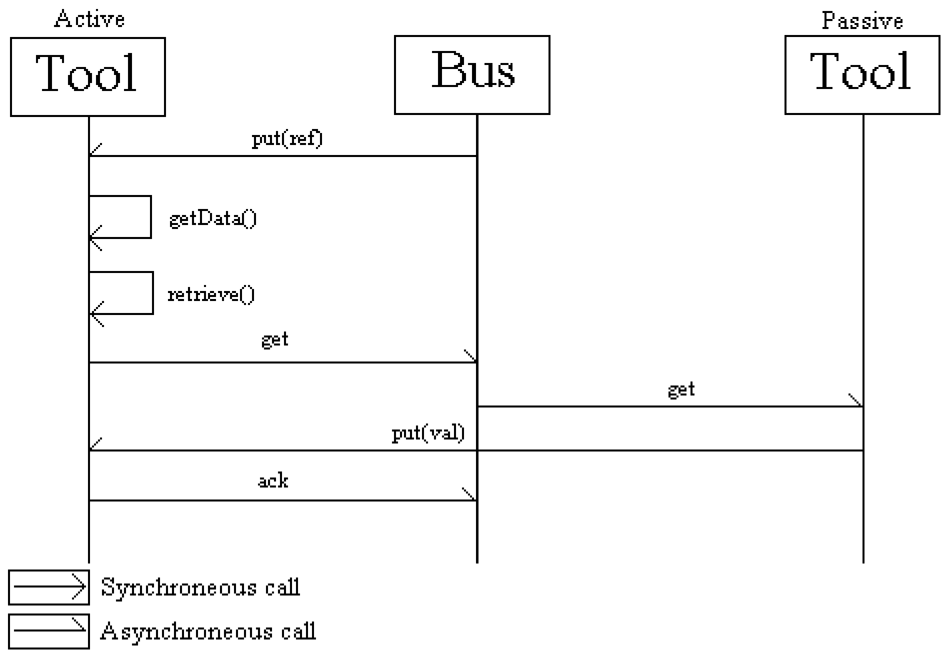

Figure 4: Sequence, active <- passive.

Passive – passive.
If both the requesting tool and the tool that is in possesion of the value are unreachable, we do have a problem. In this case we need to fall back to the regular method of communication. In addition this will also introduce some overhead related to intertool communication. This situation is very undesirable, since it will prevent any of the optimalizations to function for these requests.

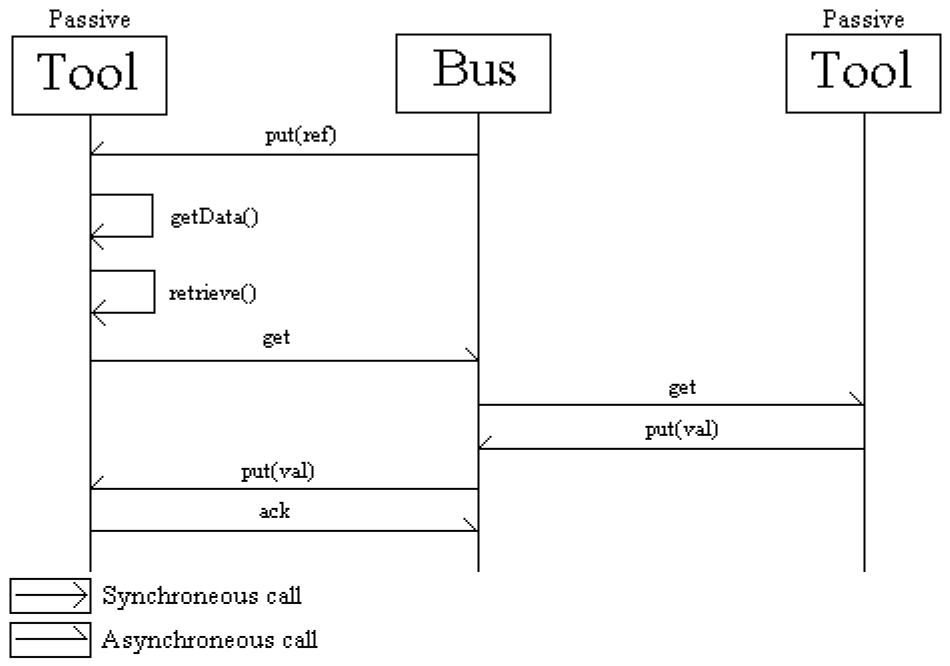

Figure 5: Sequence, passive – passive.

Finalization

Whenever a variable becomes unreachable on the ToolBus level a FIN message will be sent to the tool that is in possesion of the value, to notify it that it can no longer be requested by other tools. This way the tool knows it can collect the value at its own discretion.

Termination

To initiate the termination of a connection, an END message must be send. Upon reception of the END message, the remote side of the connection will be closed.

Additional improvements

Besides the improvements that are related to solving the main problem, several additional improvements were made. Three of them are worth noting in particular:

- A more advanced method of term serialization.
- A constant pool for received variables.
- Utilization of multi-processor / core systems.

Term serialization

During the implementation of the prototype we discovered that (de-) serialization was a major issue. To send a message we needed to serialize the entire object, which resulted in a string, in turn converting that into bytes that would be copied into a buffer. The result was that we needed four to five times as much memory as the size of the object that we wanted to send. This is also, more or less, the case in the current implementation that uses ATerms. This inefficiency is absolutely unacceptable.

Because one of the main objectives was to conserve memory, a more efficient way of serializing and de-serializing objects was needed. To do so we needed to go back to raw bytes, instead of using strings. This is also beneficial in terms of space. For example, to represent an integer you only need four bytes instead of a string containing numbers, which could be several times as large. Sending and receiving these kinds of objects would be easier and faster too, since we have a byte representation of the object and can directly send / receive it; we do not have to perform any conversions. Besides this we also wanted to be able to access any (up to the byte specific) part of the objects byte representation, allowing us to send the object in chunks. This has the additional advantage that it enables us to reuse the buffers for sending and receiving objects.

Ultimately this resulted in memory usage equal the size of the object we are serializing plus the size of the buffer we are using for sending / receiving. In terms of performance it is a lot faster as well. Compared to serializing and de-serializing a term, it is several times faster than an ATerm. And it should scale linearly with respect to object and tree size too. The price we have to pay however is complexity. Additionally, to improve serialization speed, the content of the objects is internally stored as a byte array; this causes the access to the content of the constructed objects to be mainly limited to by-value, this should be changed to improve runtime performance.

The constant pool

The constant pool is a minor improvement; it will however provide a noticeable improvement in performance when there are several tools running on the same virtual machine. Due to the fact that the value is cached inside a variable and thus only retrieved once. Sending the same variable to more than one tool that is running on the same VM will thus be more efficient. Additionally when the tool that is in possession of the value of the variable is running on the same virtual machine it will retrieve it directly instead of through a socket. Especially for applications that are running on the same machine, this will yield a more than fair performance increase.

Concurrency

The prototype is also able to utilize multi-processor and multi-core systems. Instead of handling everything sequentially, like the current implementation, the sending and receiving of messages can be done by a specifiable amount of concurrently running threads. Everything else, like the process logic, is also running in a separate thread. This way we can distribute the load among several processors / cores, ultimately improving performance. Because of the concurrency associated with transportation we can now both use the up- and down-stream capacity to it's fullest. Previously it was not possible to send and receive at the same time. Nor can we be limited by one tool with a slow connection, which could cause a slowdown in the execution of the ToolBus. The trade-off is added complexity, because more advanced locking mechanisms are required to ensure thread-safety. In this case it is worth the effort, because there are numerous advantages, as stated above.

The design

To be able to execute benchmarks and to prove the solution is viable for usage in the Next Generation ToolBus a prototype was made. This prototype is similar to the current ToolBus implementation and is fully functional apart from the fact that the process logic is hard coded for demo purposes. Here we will take a look at the design, notable things about its implementation and discuss how to merge the technology featured in the prototype with the current ToolBus implementation.

The design

Figure 6 globally sketches the design of the prototype. As you may have noticed both the 'client' and the 'server' side look similar. They both have the same structure; a data handler is connected with an IO handler, which is responsible for sending and receiving data. This standardization was done, because it should not matter what you communicate with and how this communication is handled, the only thing that is different is what needs to be done with the data.

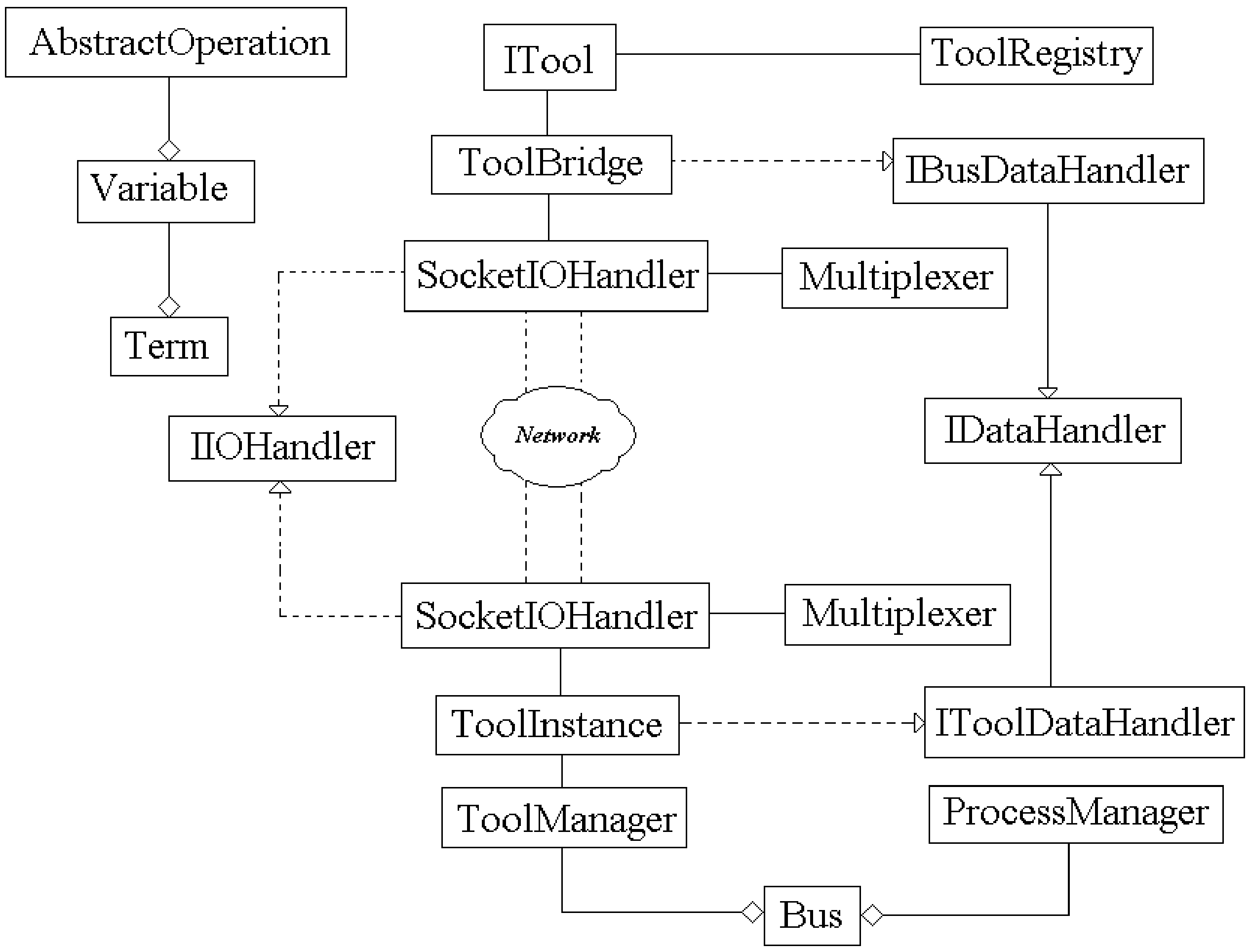

Figure 6: A global sketch of the design.

The communication (*figure 7*) can currently be done either through a socket or directly by means of Java method calls. This is dependent on the type of IO handler that is used. Alternate methods of communication can thus easily be added. The data that is exchanged is encapsulated in an 'operation'. This operation supplies facilities for serializing data (term serialization is discussed in the additional improvements section previously in this chapter). Each operation represents a package. The content of these packages is discussed previously in this chapter, in the protocol section. Usually the operation will contain a variable or a reference to a variable; the variable will in turn hold a term or information about where to find that term.

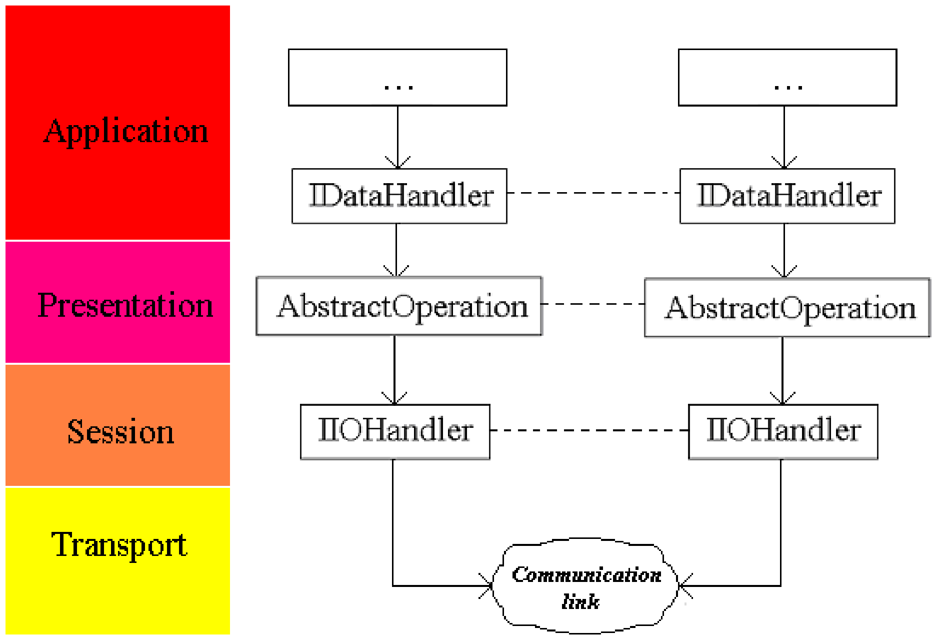

Figure 7: Communication layers.

Inter tool communication

Figure 8 shows the design of the inter tool communication part. Note that it looks very similar to the design shown in *figure 6*. Another thing that can be noted is that the inter tool data handler has a relation with the tool registry and not with the tool bridge. The reason for this is that we only need one ‘entry point’ per virtual machine to enable the exchanging of objects between tools; starting an ‘inter tool communication service’ for every running tool is not necessary.

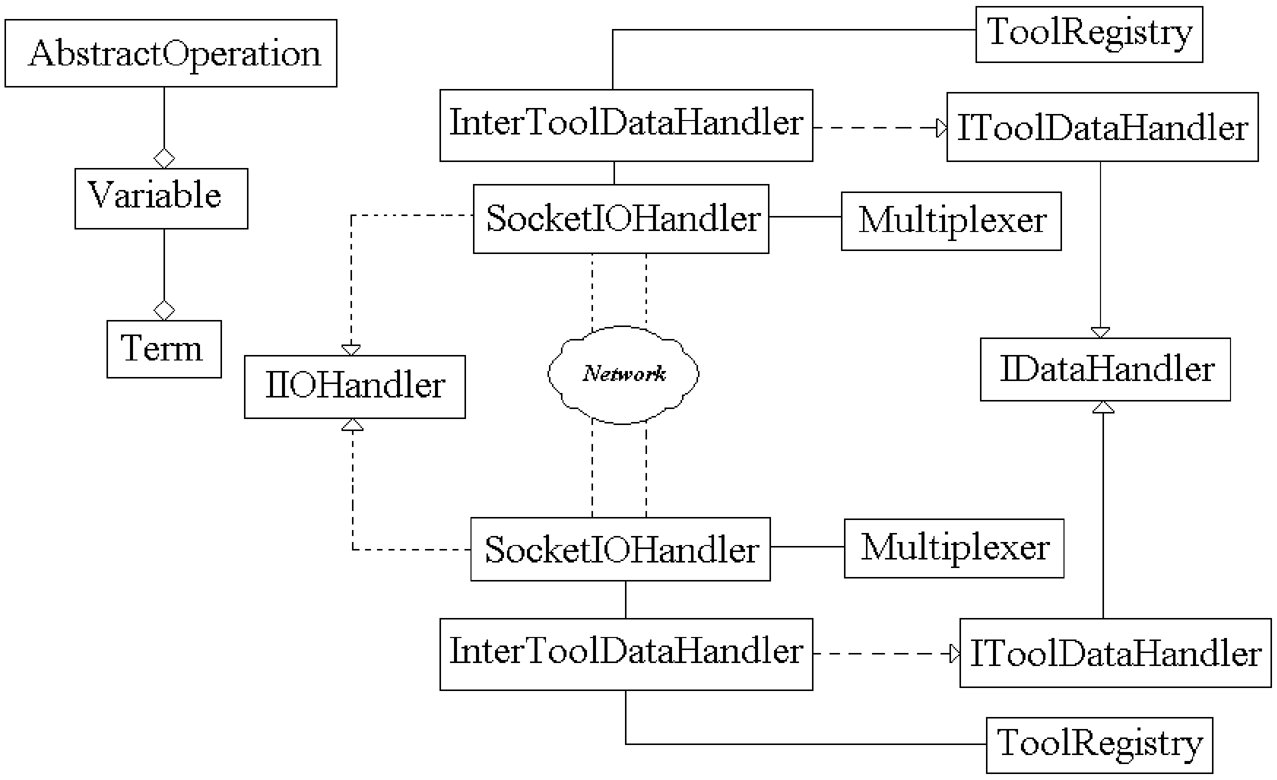

Figure 8: Inter tool communication design.

Implementation

For network communication NIO was used, which gave us the possibility of multiplexing. This meant that we could use one thread to monitor all the connections. If there is data that is ready to be received we are notified. But having only one thread running to handle all communication is not very efficient, since that would limit us to reading from or writing to only one socket at a time. That is why we decided to add two thread pools. This allows us to read and write to multiple sockets simultaneously, utilizing the available resources more efficiently. Additionally it will significantly improve performance on multi-processor and multi-core systems.

Another notable thing about the implementation is that all the tools that are running in the same virtual machine are registered at the ToolRegistry. In certain cases this allows us to exchange data directly between tools through Java method calls instead of transferring it across a socket. Even if the tool is communicating with the ToolBus through a SocketIOHandler instead of a DirectIOHandler. Copying a pointer is naturally more efficient than serializing and transmitting an object.

Merging

If the technology presented in the prototype is to be incorporated in the current ToolBus implementation certain changes will need to be made. I propose to replace the entire transport part by that of the prototype, since it is almost fully functional. Entirely writing a new implementation based on the proposal presented in this thesis should not be necessary. The most problematic part will be related to the process logic. The prototype does not have fully functional process logic, it is hard coded for demo / benchmark purposes. So there is a 'conflict' around the ProcessManager that will need to be resolved. This will be the main problem on the ToolBus side. On the client side the ToolBridge will need to be replaced and probably the entire ToolBus adapter with it, since fundamental changes are necessary to enable inter tool communication. The interface that is generated for every individual tool must be derived from the ITool interface. So this will also require changes to be made to the application that generates the interface for tools.

There is however one risk associated with the reusing of the prototype; the knowledge about its implementation might no longer be available after the end of this project.

Metrics

And finally some numbers that will give some impression about the amount of work that was put into the construction of the prototype.

Total Physical Source Lines of Code (SLOC)	= 3,975
Development Effort Estimate, Person-Years (Person-Months)	= 0.85 (10.22)
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))	
Schedule Estimate, Years (Months)	= 0.50 (6.05)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))	
Estimated Average Number of Developers (Effort/Schedule)	= 1.69
Total Estimated Cost to Develop	= $ 115,066
(average salary = $56,286/year, overhead = 2.40).	

The above statistics were generated by SLOCCount, Copyright (C) 2001-2004 David A. Wheeler.

Lines of code (comments, JavaDoc and white space included)	= 16327
McCabe cyclomatic complexity	= 1.598

Results

To determine the improvement compared to the current situation a series of tests were executed. In these tests a series of objects of a certain size were consecutively sent from one tool to one of more other tools. These tests were repeated with different object sizes to enable us to determine how inter tool communication scales with object size. Each of these tests was run at least ten times, to increase precision. However some deviations are always present, since there are factors involved we cannot control, like:

- Interference with other processes, regarding available resources.
- The scheduling policy of the operating system and the virtual machine.
- Interference from the garbage collector, which can decide to suspend the application briefly.

Especially in the tests concerning small values these factors had a noticeable influence.

The following tests where executed:

- Memory benchmarks.
- Throughput benchmarks.
- (De-) serialization speed benchmarks.

These tests when executed on a machine with the following configuration:

Processor	AMD Athlon 64 3500+
Memory	1GB DDR400 dual channel
Operating system	Linux: Fedora core 5
JDK version	Sun JDK 1.5.0
Garbage collection algorithm	Concurrent low pause collector

As stated above, we used the concurrent low pause collector as garbage collection algorithm. Even though we are using a single processor system to run the tests on we felt that due to the soft-realtime nature of the ToolBus this was the better choice. This type of garbage collection algorithm will most probably not increase performance on a single processor system, because of its associated overhead; it does however decrease the garbage collection pause times and allows the execution of the ToolBus to continue during a large part of the collection.

No other non-vital processes where running during the tests. Before every separate measurement all the applications, including the ToolBus, were restarted to ensure similar benchmark conditions. The memory measurements where done using the JConsole, which is shipped with Sun JDK 1.5 and up. The custom made test applications used for the throughput and serialization benchmarks both printed the 'used time' in milliseconds in the console after completion of the test.

Memory benchmarks

The memory benchmarks will enable us to compare the relative gain in memory usage efficiency. As will become evident from the results that are presented in the table shown below; the memory usage of the ToolBus will remain constant, independent of the size of the object that is being sent since only a reference is passed. Note that the memory usage of the sending tool increases, since it will need to hold the value of an object until it has become unreachable at the ToolBus level. This is a trade-off we are willing to make, since the values of the objects will no longer be concentrated at the ToolBus, but are now distributed among the tools that are 'sending' these objects. This is exactly what we wanted to achieve.

It is understandable that the memory usage of the sending tool is higher when inter tool communication is used to transmit objects, as explained above. In the benchmark that we are running it will need to hold both the value of previous object that was sent as the newly generated one in memory. This is because the previously generated object will not become unreachable until all the references to it (within the ToolBus) are gone. In this particular case this will not happen until the next object is sent to the ToolBus.

Something unexpectedly did however happen; the memory usage of the receiving tool increased when the data was sent through the ToolBus. The occurrence of this phenomenon is strange, since only the method of communication is different. Instead we expected the memory usage to be slightly less, since the receiving tool is not using the tool inter communication part and thus less objects are being allocated. A possible explanation for this is, because in total more system resources are used (both in terms of memory usage as processing power) the relative speed at which objects are created is higher and the garbage collector will relatively receive less scheduling time (note that the garbage collector is in this case partially running concurrently with the execution of the application, as discussed previously in this chapter). In other words, the most probably explanation is that it is caused by resource contention between different processes.

Another notable thing that is worth mentioning is the usage of soft references for the constant pool. When soft references are used the application will use all the available memory for caching objects before making them subject to collection. By default weak references are used, but the option to use soft references is present, so it can be enabled if this type of caching behavior is preferable. This can be very profitable in case the same objects are being sent over and over; if an object is already present in the constant pool it does not have to be send again, thus it conserves network bandwidth.

	Inter tool communication				Through the ToolBus (current default)			
	ToolBus	Sending tool	Receiving tool	Total	ToolBus	Sending tool	Receiving tool	Total
Idle	450KB	600KB	600KB	1650KB	450KB	600KB	600KB	1650KB
5M	1-5MB	20-25MB	5-10MB	26-40MB	10-15MB	10-15MB	10-15MB	30-45MB
5M (soft)	1-5MB	20-25MB	5-MaxHeap	26-MaxHeap	10-15MB	10-15MB	10-MaxHeap	30-MaxHeap
10M	1-5MB	60-65MB	10-15MB	71-85MB	20-25MB	30-40MB	20-25MB	70-90MB
20M	1-5MB	80MB	20-25MB	101-110MB	40-65MB	40-80MB	40-65MB	120-210MB
50MB	1-5MB	150-200MB	55MB	206-256MB	100MB	100-150MB	100MB	300-350MB

As we can see in the table above. The total amount of memory that is used will decrease. Although the gain is not so significant when only small values are sent; it becomes clearly noticeable when larger values are used. Additionally, because the object values are distributed across the sending tools and are no longer concentrated at the ToolBus, the system becomes much more scalable. This was our main goal.

Throughput benchmarks

We executed two types of throughput tests. One test with one sending tool and one receiving tool and another with one sending and two receiving tools. This will give us some idea about how it would scale when sending to multiple Tools. We executed the tests with values of different sizes and made a comparison between the regular method of communication and inter tool communication.

We are expecting to see a significant increase in performance in both tests. Since the data has to be sent one time less in total, the performance gain should become less noticeable if we add more receiving Tools. We expect inter tool communication to be twice as fast in the first test and yield an increase of about fifty percent in the second case.

Using inter tool communication to transport an object will however introduce some overhead, because extra control messages have to be sent. This makes it interesting to find out at what message size it would be faster to send the message directly through the ToolBus. Unfortunately when executing the benchmarks with very small messages, the results seemed to become more unreliable, even taking an average from a large number of runs did not completely solve this. The same goes for the 'dips' of 31.5% and 24.7% in the first test, which could not be explained and are most probably caused by external influences. Besides that, the optimal value of the 'threshold' will be dependent on the available bandwidth and latency of the connected tools as well. For this reason we decided to leave it up to the user to make decisions about the threshold.

1 sending tool 1 receiving tool	Inter tool communication	Through the ToolBus (current default)	Gain in terms of percent
128KB	68.6 ms	66.8 ms	-2.6%
512KB	90.8 ms	148.3 ms	63.3%
1MB	122.9 ms	222.9 ms	81.4%
5MB	302.8 ms	398.3 ms	31.5%
10MB	442.9 ms	705.6 ms	59.3%
20MB	670.7 ms	1051.5 ms	56.8%
50MB	1854.5 ms	2312 ms	24.7%

When we look at the second test, the results match our expectations nearly perfectly. In contrast to the first test, in which we expected to see a greater increase. While the gain certainly is noticeable, it does not come close to the hundred percent increase we expected to achieve. Additionally, the tests seemed to be noticeably influenced by external factors, because the gain does not scale in any particular way. Even repeating these test several times and drawing an average from the results did not stabilize these numbers.

1 sending tool 2 receiving tools	Inter tool communication	Through the ToolBus (current default)	Gain in terms of percent
128KB	119.1 ms	103.65 ms	-13%
512KB	145.95 ms	182.6 ms	25.1%
1MB	213.15 ms	233.95 ms	9.6%
5MB	383.85 ms	525.55 ms	36.9%
10MB	603 ms	886.45 ms	47%
20MB	915.5 ms	1313.55 ms	43.5%
50MB	2026.5 ms	2864.1 ms	41.4%

In any case, the performance gain is clear. The only exceptions are the tests with small values. But as already stated above, the results of the tests with relatively small values are not completely reliable, since they are more prone to deviations caused by external influences. We expect that the performance gain will be even more significant if we would run the same test in a distributed environment. The reason for this is that the measurements in this case will suffer from a greater influence from the connection speed and less from the overhead associated with inter tool communication. This should also make the results scale more linearly with the amount of data that is being sent.

Serialization benchmarks

Besides the benchmarks related to the ToolBus it might also be interesting to look at the performance improvement that was made with regards to the serialization of Terms. The current implementation of the ToolBus works with ATerms. Because these were insufficient for usage in the prototype we made a new kind of 'term'. This is discussed in more detail in the previous chapter. Both the tests serialized and de-serialized a term (or tree of terms) one hundred times in a row. The results are impressive; the serializable object is much faster in both cases. This gain is mainly caused by the fact that no conversions need to be executed, preventing any garbage from being created and improving memory efficiency considerably.

Serializing and de-serializing	Serializable object	ATerm (current default)
A single term of 5 MB	2482 ms	107851 ms
A list containing 3 terms of 1 MB	2134 ms	58031 ms

Evaluation

The main goal of this project was to improve the scalability of the ToolBus by conserving memory and bandwidth. As became evident in the previous chapter both memory usage and transportation times where cut considerably. The memory usage of the ToolBus stayed between one and five megabytes, even when a large number of tools would be connected this will not increase much. Only small objects and references are present at the ToolBus that will usually not exceed more then a couple of kilobytes, depending on the threshold.

Concerning garbage collection, every remotely accessible object will be instantly notified when it has become unreachable at the ToolBus level. We can be sure all objects will be collected in a timely manner and only when they have become truly unreachable. How this is handled is discussed in the 'project execution' chapter.

Even after implementing these fundamental changes to the ToolBus, semantically it should still behave the same.

Naturally we ran into some problems during this project, as is to be expected. Fortunately these problems did not take long to solve. The decision to implement the prototype as a separate application proved to be a good one. We would have run into serious problems with the ATerm implementation if we had done so, since it did not supply the required functionality. It was lacking in terms of memory efficiency with regards to serialization and it did not support the traversing over signatures. It saved us a lot of valuable time. The down side of this choice was however that we now have two different ToolBus and 'term' implementations, which will have to be merged.

To summarize, we proved that the presented solution is indeed working and is applicable for usage in the current ToolBus implementation. The major accomplishments of this project are:

- Drastically lowering the memory usage of the ToolBus.
- Increasing the throughput significantly.
- Improving (de-) serialization speed by a multiple.
- Adding multi-processor / core support.

Ultimately this will lead to a faster and more scalable system as is proven in the prototype.

However some changes have to be made to the ATerm library to make these improvements possible. In particular:

- Traversing signatures, since we do not have (and do not want) the value at the ToolBus level, but still need to be able to access a certain node in a tree.
- More advanced serialization; we want to be able to serialize particular parts of a tree / term, not just get one big BLOB. A suggestion about how to incorporate this in the current ATerm implementation is done in Appendix B.

Future work

Due to the short amount of time that was available for this project, not everything could be implemented. Besides that, there are also some additional improvements that could be made. We will walk through these here.

Retrieving sub terms

Something, rather important, that has not been implemented yet is the ability to retrieve a sub term. What is meant by this is that on the ToolBus level a sub term could be extracted from a larger tree and sent to a Tool. When this tool wants to retrieve this value it will need to send a request including the variable identifier of the root term of the tree and the identifier of the sub term we want to retrieve. Because we are only matching and traversing signatures and not values on the ToolBus level we will need to relate the different 'nodes' in these signatures to the identifiers of the terms associated with them. Additionally we will need to retain the entire tree until it, including all of its sub terms, has become unreachable. Some optimizations, concerning this, might be implemented later.

Firewall support

As discussed previously in this chapter, support for firewalls and routers is required to increase the amount of situations in which inter tool communication can function. The idea is ready, but unimplemented. Detecting reachability can be done by checking whether or not a connection has been refused. If it is, we can assume we are dealing with a passive Tool. Although this will work, a more advanced way of detecting and 'remembering' passiveness would be beneficial, because we can then better adjust the method of communication to the situation, limiting communication overhead. For example, why would you keep trying to connect to a tool if you already know it is unreachable and you need to issue the request through the ToolBus?

Streams

Streams would be beneficial in case we need to send large objects through the ToolBus, in case inter tool communication is unavailable, due to firewall restrictions for example. When we need to send a large value across the ToolBus, opening a stream to tunnel the data through would be better than sending the entire BLOB at once. Multicast mechanisms could also be constructed based on this, functioning similar to the currently implemented subscribe function. The specifics about what an implementation would look like are unknown at this time.

Security

Currently there is no security whatsoever. Everyone can connect with the ToolBus. Using SSL sockets instead of the regular ones would be an improvement. The performance penalies this will introduce will most probably not be a concern.

Reconnecting Tools

Restoring a connection with an unexpectedly disconnected tool is currently not possible. In the implemented prototype it is however detectable, it just is not being handled as of yet. One option would be, queueing all the messages that should be sent to the tool and sent them as soon as it reconnects. Reconnecting is the tool's responsibility, since tools can not accept connections. Another option would be to stall the processes associated with the disconnected tool and re-register them as soon as it reconnects. This can only be done when the functions the tool offers are completely re-entrant. In both cases it would be wise to use leases, we do not want to hold everything indefinitely. The prototype currently handles it in the following way; it destroys the tool instance and the processes associated with it in an attempt to minimize the damage, hopefully localizing the impact to the disconnected tool only.

Bibliography

1. J.A.Bergstra, Paul Klint. The Discrete Time ToolBus. Technical Report P9502, Programming Research Group, University of Amsterdam, 1994.

2. J.A.Bergstra, Paul Klint. The Discrete Time ToolBus: A Software Coordination Architecture. Science of Computer Programming 31:205-229, 1998.

3. Hayco de Jong, Paul Klint. ToolBus: The Next Generation. Formal Methods for Components and Objects, LNCS 2852, 2003.

4. M.G.J. van den Brand. H.A. de Jong, P. Klint. P.A. Olivier. Efficient Annotated Terms. Software -– Practice & Experience 30:259-291.

5. Richard Jones, Rafael Lins. Garbage Collection: Algorithms For Automatic Dynamic Memory management. John Wiley & Sons Ltd. 1996.

6. Paul Watson, Ian Watson. An Efficient Garbage Collection Scheme For Parallel Computer Architectures. Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe, pages 432 – 443, 1987.

7. Benjamin Goldberg. Generational Reference Counting: A Reduced Communication Distributed Storage Reclamation Scheme. Conference on Programming Language Design and Implementation, pages 313 – 321, 1989.

8. José M. Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. Proceedings on Parallel Architectures and Languages Europe: Volume I: Parallel Architectures and Algorithms, pages 150 – 165, 1991.

9. Bernard Lang, Christian Queinnect, José Piquer. Garbage Collecting The World. Symposium on Principles of Programming Languages, pages 39 – 50, 1992.

10. José Piquer. Indirect Distributed Garbage Collection: Handling Object Migration. Transactions on Programming Languages and Systems, Volume 18, pages 615 – 647, 1996.

11. New I/O Apis. 2002. http://java.sun.com/j2se/1.4.2/docs/guide/nio/.

12. Monica Pawlan. Reference Objects and Garbage Collection, August 1998. http://java.sun.com/developer/technicalArticles/ALT/RefObj/.

13. Doug Lea. Synchronization and the Java Memory model, July 2000. http://gee.cs.oswego.edu/dl/cpj/jmm.html.

14. Bill Pugh. The "Double-Checked Locking is Broken" Declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

Appendix A: Implementation problems

During the implementation of the prototype we ran into a couple of rather annoying problems. These problems where mainly related to design issues and shortcomings of the virtual machine. In this chapter we will discuss these problems, hopefully preventing other people from running in to them.

Multiplexer

The non-blocking IO supplied by the NIO framework enables the construction of highly scalable network applications. The idea is to use a selection mechanism to detect whether channels are ready to read from or write to. This prevents you from having to start a separate thread to service every socket.

However, when you register a socket for the write operation with a selector you will probably end up with 100% CPU cycle consumption. The reason for this is that the `select()` call will return immediately, because the channel is ready to be written to, since the network output buffer is not filled completely. This will happen even if you do not have anything to write. The problem with this behavior is that system administrator probably will not like you very much when you are needlessly consuming the maximum amount of resources available (and we do not want that to happen, now do we).

The solution is to only register for writing when you actually have something to write and deregister when you are done. Sounds easy enough, but it is not. Simply because the design for the selector is flawed. Consider the following piece of code:

```
while(running){
   selector.select();

   Set keys = selector.selectedKeys();
   Iterator keyIterator = keys.iterator();

   while(keyIterator.hasNext()){
      SelectionKey key = (SelectionKey)keysIterator.next();
      keyIterator.remove();

      if(key.isReadable()){
         read(key);
      }
      if(key.isWritable()){
         write(key);
      }
   }
}
```

Registering for the write operation can be done by calling the `SelectableChannel.register(selector, int, Object)` method. This method synchronizes on the selector. There is however a 'minor issue'; the monitor on the selector is being held by the thread that is blocked on the `select()` call. Calling `wakeup()` on the selector, which causes the `select()` call to return immediately and release the monitor, will not work either due to the simple fact that if there is nothing to do and the thread will almost instantly be blocked in the `select()` call again. The chance that the thread requesting the registration for the channel will be scheduled within this limited time frame is close to zero.

The solution for this problem is to introduce a 'barrier', which will prevent the `select()` call from being executed after the `wakeup()` call. This could, for example, look like this:

```
while(running){
   selector.select();

   Set keys = selector.selectedKeys();
   Iterator keyIterator = keys.iterator();

   while(keyIterator.hasNext()){
      SelectionKey key = (SelectionKey)keysIterator.next();
      keyIterator.remove();

      if(key.isReadable()){
         read(key);
      }
      if(key.isWritable()){
         write(key);
      }
   }

   synchronize(selectionPreventionLock){
      // This is the barrier
   }
}
```

If we synchronize on the lock used for the barrier before calling `wakeup()` the `select()` can not be executed before we release it, giving us the ability to register the channel, like this:

```
synchronize(selectionPreventionLock){
   selector.wakup();
   socketChannel.register(selector, SelectionKeys.OP_READ | SelectionKeys.OP_WRITE,
null);
}
```

Registering from inside the same thread will not be a problem of course. The selector was obviously not designed with concurrency in mind.

Concurrency

Programming a multi threaded application can become very complicated and mistakes are easily made if you do not fully understand what you are doing. The most annoying thing about bugs that are the result of a concurrency problem is that they are extremely hard to find and debug. For one they occur very rarely (if at all) and secondly adding debug statements might in some cases introduce a slight delay preventing the bug to be reproduced. Adding `System.out` statements will not work either because they cause a monitor release after writing to it (it's synchronized) and thus influence the semantics of your program, complication things even more.

Additionally Sun did not make it easier for programmers either, since the Java Memory Model in JDK 1.4 and earlier is fundamentally broken. It should however be better in JDK 1.5.

One of the major problems is that the compiler can move the statements in your code around as long as the as-if-serial semantics are maintained. Under this condition the following re-orderings are valid:

- Move around statements inside a method.
- Move around statements within a synchronized block.
- Move a statement from outside, into a synchronized block (but not the other way around).
- Move around a statement involving a variable declared as being volatile (up to JDK 1.4, but not in JDK 1.5).

The compiler can also do additional re-orderings after one or more calls to other methods have been inline expanded. This can sometimes lead to unexpected behavior, like in the example below. Consider the following code:

```
int i = 0;
int j = 0;
int x = 0;
int y = 0;
Object a = new Object();
Object b = new Object();
```

Thread 1

```
synchronized(a){
   x = 1;
}
y = 1;
```

Thread 2

```
synchronized(b){
   j = y;
}
i = x;
```

The compiler can validly transform this code into this:

```
int i = 0;
int j = 0;
int x = 0;
int y = 0;
Object a = new Object();
Object b = new Object();
```

Thread 1

```
synchronized(a){
   y = 1;
   x = 1;
}
```

Thread 2

```
synchronized(b){
   i = x;
   j = y;
}
```

Which can cause i to become 0 and j to become 1, since thread 1 and 2 are synchronizing on different objects and the compiler can freely reorder the content of both methods. Although they still do exactly the same as before, the outcome can be different.

A more detailed description about compiler based reordering can be found here [13].

Besides the compiler ‘optimizations’, the processor can influence the order in which statements are executed and the memory system the order in which there results are written into the main memory. This is mainly a problem on multi processor / core systems, provided that every processing unit has its own cache.

Processors can for example execute an atomic operation in between a non-atomic one. For example a assignment to a long and an integer; the first half of the long might be written first, then the entire 32 bits of the integer and afterwards the second half of the long (possibly causing strange behavior by itself, since it can cause a half updated value to be read). This will cause the second assignment to be done before the first, being a kind of reordering as well.

The order in which the processor synchronizes updated values with the main memory is also undefined. So another thread could see an updated value of one variable, but not for a variable that was modified previously by that same thread. But the other way around is possible too; a processor could still have a value of a certain variable in its cache, even if the value in the main memory is updated it may not be reloaded, causing the stale value to be used.

This is also the case with variables that are declared to be volatile. For these variables are no guaranties with respect to the reordering of reads and writes to the main memory, nor on compiler optimalizations. It will only guarantee that the reads and writes will be done directly in main memory, preventing it from being cached in a register so every thread will see the most up-to-date value. Under these conditions volatile was practically useless, so they 'fixed' them in JDK 1.5. Reads to volatile fields will now behave as a monitor acquire and writes as a monitor release. Additionally statements involving variables declared as volatile will restrict any compiler optimalization with respect to reordering. This way we can ensure that we always read the most up-to-date value, including that of the variables modified by previously executed statements.

Most of the above stated problems could be solved by means of synchronization, but not all.

An example of this is that it can appear that immutable objects change their state. This is caused if a reference is passed and used by a different thread right after (or during) the object's construction. The String class is one of the objects that suffers from this 'syndrome'. It can occur that certain fields have not been synchronized with the main memory yet when a different thread accesses it. This causes the newly created String to appear to be incorrect at first (only contain default values for example) and change its state to the correct value a moment later. All immutable objects suffer from this problem. This should be fixed in JDK 1.5 as well. In this JDK final variables should behave similar to volatile variables inside the body of the constructor; before the constructor is executed all final fields that are used should be reloaded (weak acquire) and when it completes all the updated final variables should be synchronized with the main memory (weak release), insuring that all threads see the properly initialized object. An additional benefit associated with these semantics is that the value of the final field never has to be synchronized with the main memory again from that point on, since it is properly initialized and will not change its value. Note that this only works for immutable objects, regular objects will still suffer from this problem. You will need to supply proper synchronization, if you want to pass these objects between different threads.

The most infamous example of a concurrency problem is Double-Checked Locking [14]. It is impossible to get it working in Java, even under the semantics in JDK 1.5.

So there are enough things that can go wrong, that you need to take into account.

Bit shifts

One other rather annoying thing I ran into is that bit shifts do not always behave as expected. Although they are working correctly and are documented. The following thing does not 'work':

```
byte aByte = 1;
long aLong = aByte << 33;
```

This will result in 2. The cause of this is that the variable we are shifting will be promoted from a byte to an integer. So shifting a byte more than 31 bits does not result in a long as you might expect, but in an integer with 2 as value, since we caused an overflow. Casting the byte to a long before shifting solves this problem. Like this:

```
byte aByte = 1;
long aLong = ((long)aByte) << 33;
```

It can be very frustrating if you do not know this.

NIO buffers

There is one bug related to NIO that Sun, seemingly, does not want to acknowledge (Sun bug: 4879883). Its state was set to: Closed, fixed at 08-23-2003. Yeah right, it is not!

The problem is the following: when reading from an external source (like a socket or file) the VM allocates a buffer in 'direct' (native) memory. This buffer is filled with the data that is being received and which is, in turn, copied over to a 'java NIO buffer' in the heap of your application. The problem is that the buffers in direct memory are not reclaimed in a timely matter, if you are reading a lot of data (50MB or more) it will throw an out of memory exception.

Sun's 'solution' was to increase the amount of memory the VM is allowed to allocate direct buffers in. I even tried setting it to high values (256MB instead of the default 64MB), but even then receiving 50MB was impossible. The only way to get it working is to manually allocate direct buffers and reuse them. This way it will never have to be reclaimed. I/O operations will be done directly on these buffers, instead of copying them over to 'java NIO buffers'. This will prevent the VM from allocating direct buffers itself; ultimately solving the problem.

And additional gain was improved and more stable performance. There are several reasons for this:

- I/O operations on direct buffers consume less CPU cycles then operations on 'java NIO buffers'.
- The content of the buffer now only needs to be copied once instead of twice (direct -> target instead of direct -> java NIO buffer -> target).
- Allocating a (fitting) block of contiguous native memory can be quite costly.

Appendix B: ATerm serialization improvement

As discussed before, the performance on the Java implementation of the ATerm library is pretty poor and can be improved considerably. Here we will do a (global) proposal about how to achieve this increase in performance.

The problem

There are two main problems. The first is memory usage and the second serialization and de-serialization speed. Currently, if you want to serialize an ATerm you will need more than four times the amount of memory as the size of the term. This will even increase more when we use the `toString()` method on a term to obtain a serialized representation, since that does not support sub-term sharing. The cause of this absurd amount of memory usage is because temporary data is constantly being copied back and forth. This 'copying' is one of the sources of the poor performance. Additionally this causes a lot of garbage to be created, meaning the garbage collector will be making overtime trying to reclaim all of it. Although these temporary objects are all 'short lived' and thus can be collected in a 'minor collection', it causes an unnecessary slowdown. The second performance problem is caused by the fact that every character is checked to see if it needs to be escaped. This is needed because the serialized representation of an ATerm contains certain 'control characters'.

Apart from this the ATerm library is not able to return a serialized representation in a format that is ready to be transmitted, for example across a socket. You will need to do an additional conversion (from String to byte[]). Because the Next Generation ToolBus is using NIO, it might be better to return byte arrays or ByteBuffers instead.

The solution

First of all there is one limitation, we do not want to make any fundamental changes to the ATerm library that would endanger backwards compatibility. Thus we need to make an augmentation that supplies a more efficient method of (de-) serialization. What we propose to do to increase memory efficiency, (de-) serialization speed and ensure a seamless connection with the Next Generation ToolBus is the following:

First a few facts:

- We are hardly ever able to send / write an entire term at once.
- Buffers are usually reused.
- Terms are serialized sequentially.

With these things in mind, it might be best to create an interface that is able to fill a ByteBuffer with the serialized representation of a term. The amount of bytes written into it depends on the size of the buffer and it's 'limit'. The next time a buffer is passed to this interface it will continue serializing where it was left. This can be continued until the entire term has been serialized. In case the buffer that was passed cannot be filled completely, the limit should be set to indicate how many bytes where written into it. Keep in mind that data must be written into the buffers directly without creating any temporary objects (or if it can not be avoided, as little as possible). De-serializing will go in much the same way. One change in the design is however required to enable support for de-serialization; either the constructors of the terms must be given package access, or every term should be able to (de-) serialize itself. This is because terms can be solely constructed by parsing a string in the current implementation.

The format

Of course a new format is needed as well. This is shown in the table below (related fields are colored similarly).

Field	Size	Description
Is shared	1 byte	Specifies if a term is coming or a value
Term index	4 bytes	In case of a shared sub term this contains a 'pointer' to the value of the term
Name length	1 byte	Specifies the length of the name
Name	2^8 bytes limit	The name of the term
Content count	4 bytes	Specifies the number of content nodes
Sub-term count	4 bytes	Specifies the number of sub-terms
Annotation count	4 bytes	Specifies the number of annotations
Content node length	4 bytes	Specifies the length of the content node
Content node	2^32 bytes limit	One of pieces of content of the term. A term can contain multiple content nodes

Note that all of the 'counts' are currently four bytes (because integers are 32 bits values), if less is acceptable this should be changed. Sub-term sharing should be incorporated as well. We propose to do this by setting a flag in front of every term to indicate what is coming, a term or a 'pointer' to a term. This 'pointer' can be either a hash code, as it the case currently, or a number indicating the 'index' of the term it points to (a sort of LZW-like idea), which might be faster. It may seem that the serialized representation of terms will be slightly larger then it is currently because of the header. However, because we will no longer need 'control characters' (like: #(){}[],), taking away the need to escape characters and adding sub-term sharing, we expect them (in most cases) to be smaller then they are now.

This is an example of what a term with two equal sub terms and an annotation would look like (similar terms have the same color):

Is shared	0
Name length	8
Name	snd-data
Content count	0
Sub-term count	2
Annotation count	1
Is shared	0
Name length	4
Name	data
Content count	1
Sub-term count	0
Annotation count	0
Content node length	8
Content node	TestData
Is shared	1
Term index	2
Is shared	0
Name length	15
Name	test-annotation
Content count	1
Sub-term count	0
Annotation count	0
Content node length	21
Content node	TestAnnotationContent

Conclusion

By implementing the discussed solution, the performance of the Java ATerm implementation will increase significantly. It will also allow for a seamless connection with the Next Generation ToolBus. Memory usage will be cut by at least a factor of three to four. Serialization speed increases of two- to three thousand percent are to be expected.