Gears4Net - an Asynchronous Programming Model

Martin Saternus, Torben Weis, Sebastian Holzapfel and Arno Wacker
Distributed Systems Group
University of Duisburg-Essen
47057 Duisburg, Germany
Email: {martin.saternus | torben.weis | sebastian.holzapfel | arno.wacker }@uni-due.de

***Abstract*—We propose a programming model for scalable distributed applications. Our programming model aims at applications which have to handle a high number of concurrent network connections in parallel. Such load characteristics appear in Web 2.0 applications and in scientific network simulators. The problem originates at the I/O API provided by the operating system. Blocking I/O is easy to use, but in large scale scenarios the usage of threads and memory is excessive. Non-blocking I/O can theoretically reduce the resource consumption, but it is awkward to program. Our approach utilizes a special language feature found in Microsoft's C# and Python and combines the best of both worlds. Programming is almost as easy as with blocking I/O and the resource usage is nearly optimal. Our solution is implemented on top of Microsoft .NET or Mono/Linux respectively and does not require any changes to the operating system, which is very valuable for its practical application.**

***Keywords*-Parallel Programming; Programming Models; Concurrency**

I. Introduction

Today large scale AJAX web-applications must hold thousands of concurrent network connections and user state, which is very resource intensive. These connections are typically TCP connections which are used for HTTP. In AJAX Web 2.0 applications it is a common pattern to open a TCP connection, send an HTTP request and delay the response up to several minutes until the server wants to inform the web browser about something. As a result, the number of concurrent connections is huge (10.000 or more) while these connections are idle most of the time.

A similar problem arises when simulating and testing peer-to-peer software, for example in our peers@play project [1]. In this scenario a developer launches 1000 or more instances of our peer-to-peer software on a set of simulation servers. Each peer holds 16 to 32 TCP connections to other peers.

The easiest (but inefficient) way to implement the application-layer networking protocols is to use blocking I/O. Thus, each incoming connection is assigned a thread which executes the application protocol (i.e. HTTP or a peer-to-peer protocol such as Chord or Symstry [2]). This leads to very readable source code since you can follow the protocol line by line. The problem is that the number of connections is high (10.000 or more) and each connection is open for a long time (up to several minutes). Unfortunately, even modern systems either fail to handle > 10.000 threads at all or the memory usage is excessive.

The other alternative is to use non-blocking I/O. This way there is no need to allocate one thread for each connection. The drawback is that the resulting programming model is extremely difficult and therefore error prone. We have observed that many students fail to develop correct implementations when they are forced to use non-blocking I/O. Those projects which succeeded suffer from hard to understand source code, because the implementation of the protocols consists of a multitude of callback handlers.

Our goal was to develop a programming model which allows us to produce readable source code even for complex application protocols. At the same time, the number of threads should not be bound to the number of open connections, because threads are a scarce resource. Instead, the number of threads should be dependent on the number of CPU cores and threads should ideally not block, i.e. in an ideal case each thread is either ready or active, but never waiting. Furthermore, main memory becomes a scarce resource even when servers are equipped with 8GB or even 16GB. Thus, memory consumption should be as little as possible.

In the following we discuss the blocking and non-blocking operating system APIs in more detail and explain why threads and memory are a scarce resource. In chapter III we introduce the special language features of C# and Python that we utilize. In chapter IV we present our solution, show how to program with it and discuss the resource usage. Finally, we discuss related work and an outlook on future research.

II. Programming Models

A. Blocking I/O

On a Windows Vista machine with 32 bit and 2GB RAM the number of threads is limited by around 1400 if the stack size is set to the default of 512KB. One reason for this shortage is that the machine is running out of address space. Tests on Windows Server 2008 with 64 bit and 6GB RAM allowed us to start up to 9000 concurrent threads. However, in both cases the system becomes extremely unresponsive because the operating system was not designed for this

number of threads. As a rule of thumb, the memory usage can be approximated by

$$\text{Memory} := \#\text{Threads} \cdot \text{StackSize} = \#\text{Connections} \cdot \text{StackSize}$$

When launching one thread per connection, we can easily calculate that 10.000 connections and 512KB Stack results in more than 5GB of stack. This is only for stacks, i.e. no heap, no buffers for the TCP connection etc. and it gets worse with more open connections. Thus, the easy programming model (using blocking I/O) suffers from a limitation of threads and wastes memory.

B. *Non-Blocking I/O*

The asynchronous paradigm reduces the number of threads to a minimum. In theory one thread is sufficient. However, the use of non-blocking I/O increases the programming complexity and the source code is hard to understand. Actively polling more than 10.000 connections is not an option. Thus, the only viable alternative is to use the `select` operation of the socket API. When read or write is possible on any connection, select will return. This is the time to fetch a thread from the thread-pool and invoke some callback handler.

The major drawback is two-fold. First, the protocol implementation is distributed in many callback handlers. Thus, it is hard to understand the protocol by looking at the source code because one cannot immediately tell when and in which order each handler is supposed to be called. Second, there are many thread-synchronization challenges ahead, because each handler is invoked in a worker-thread. If message a is received before message b it is possible that the handler for b finishes before the handler for a because they run in concurrent threads. This will lead to out of order execution problems which would not arise when using blocking I/O. In blocking I/O the thread would wait for a request, produce a response, read a request, produce a response and so on. Thus, the use of non-blocking I/O leads to a fully asynchronous programming model which features more parallelism than intended. The developer has to come up with extra code to remove the unwanted parallelism.

At least the memory consumption is much better. We ignore the memory used to hold 10.000 connections open, since this is the same for all programming models. Furthermore, the heap usage is the same, too. The difference is the number of stacks.

$$\text{Memory} := \#\text{Threads} \cdot \text{StackSize} = \#\text{CPUCores} \cdot \text{StackSize}$$

As the above rule of thumb shows, the stack usage depends on the number of working threads which depends on the number of CPU cores. Thus, when increasing the number of connections, there is no additional stack size penalty as in the blocking case.

C. *Complex Event Detection*

Another problem that we faced in various projects is that of complex event detection. Especially in peer-to-peer protocols developers often face complex waiting conditions. For example, the protocol says to send a ping message every 10 seconds. If there is no matching pong in 60 seconds, close the connection. If a close message is received, send a bye message and close the connection.

These kinds of problems typically lead to some race conditions when detecting the event. The reason is that the timers are executed in their own thread. For example, the pong message arrives when the timer is triggered. In the end, one handler will process the pong message while another handler tries to close the connection believing it timed out.

Our approach is to make such complex event detections explicit in the programming model such that a developer can easily see what the waiting condition is. Furthermore, the processing of the waiting condition is done by our Gears4Net. Under the assumption that Gears4Net is free of race conditions, it follows that the number of race conditions in the application code will be reduced drastically.

While some aspects of Gears4Net are designed to reduce the resource consumption (memory, threads) in high performance settings, other aspects are useful even for non-high performance projects, because many sources for race conditions are removed by the use of Gears4Net. Thus, the Gears4Net API combines the benefits of all three paradigms.

III. ITERATORS IN GEARS4NET

Gears4Net was built to combine the asynchronous and non-blocking programming model with an intuitive state-machine like paradigm. To achieve this goal we have to eliminate blocking method calls and to replace asynchronous callback handlers with a new Receiver-mechanism. The elimination of blocking calls is necessary to avoid the enormous resource consumption of threads. The callback replacement is required to establish a new blocking-like wait statement.

Following the idea of non-blocking method calls in combination with wait-statements. Gears4Net executes the application source-code in a worker-thread until a wait statement occurs. The wait-condition will be added to a global waiting-condition-tree, the program counter will be saved, and the worker thread will be returned. Once an asynchronous I/O operation returns a result (this happens outside the worker thread), Gears4Net builds a message and sends it to itself. In the worker thread, these messages are dequeued. If the corresponding action meets a wait-condition, the saved program counter will be restored and Gears4Net provides a worker-thread to continue until another wait-statement occurs.

Listing 1 shows a pseudo code example of the Gears4Net programming model. Once the Execute function is called by

a worker thread, the asynchronous method SendRequestAsync will be invoked. The program counter will be saved immediately after the wait-statement in Line 3 is evaluated. When the waiting condition is satisfied, Gears4Net provides a worker-thread and continues the execution at the position of the restored program counter. In case of the given example the execution will continue in Line 4 and print the received data.

```
1   function Execute() {
2      SendRequestAsync()
3      Wait(condition)
4      Print(result)
5   }
```

Listing 1. Programming Model Idea

This programming model requires a mechanism to save and restore the program counter whenever the function must wait. Furthermore, the local variables of the function must be saved until the function is resumed. The most reasonable solution is the iterator-concept of the Microsoft .NET framework. Iterators are designed to simplify the development of custom data-structure iterations. To relieve the complexity of iterator-development Microsoft introduced the yield-statement well known from languages such as Python.

An iterator is implemented as a class-member method. The generic return type of `IEnumerator<T>` marks a method as an iterator. Whenever a yield-return-keyword occurs the iterator returns a value of type `T` and interrupts the iterator-execution. The program counter will be automatically saved by the .NET framework. A typical iterator example is presented in Listing 2. The `CallIterator`-method containing a for-each-loop iterates over the GetAuthors-method returning the authors initials.

```
1   public IEnumerator<string> GetAuthors()   {
2      field = new string[] { "ms", "tw", "sh", "aw"};
3      for (int i = 0; i < field.Length; i++)
4         yield return field[i];
5      Console.WriteLine("Leaving iterator");
6   }
7
8   public void CallIterator() {
9      foreach (string autor in GetAuthors())
10        Console.WriteLine(autor);
11  }
```

Listing 2. Programming Model Idea

When the for-each-loop invokes the iterator for the first time the program counter jumps to line 1 and starts executing the iterator's source-code until the first yield return-statement is invoked in Line 4. Thus, the string array (`field`) will be initialized and the for-loop will iterate once. The iterator yields the value of the array at index 0, saves the program counter and jumps back to line 9. The for-each-loop will move the iterator forward until it terminates. The program counter will be restored and saved in each iteration step. Thus "ms", "tw", "sh" and "aw" will be printed on the command line when the iterator terminates.

While the mechanism of saving and restoring the program counter is discussed extensively we still have to save the local variables of the iterator somewhere (e.g. `field` in the example). Disassembling the code generated by the C# compiler reveals that the C# compiler does not treat iterators as methods. Instead, the compiler generates a class for each iterator. The local variables of the iterator become class-member fields. Thus, the iterator saves its entire state in an object which in turn resides on the heap. There is no need to save and restore any stack state, which turns iterators into a stack friendly construct.

Gears4Net builds heavily on these iterators. Our approach is designed for C#, but it can be translated to other languages with a similar iterator concept, such as Python.

IV. Gears4Net Programming Model

Gears4Net is a framework for realizing highly parallel networking applications. It builds on C# and its iterator concept as discussed in the previous section. The goals of Gears4Net are twofold: a) support a high number of concurrent network connections and b) offer an easy programming model. Gears4Net is based on four core concepts: schedulers, protocols, state machines, and waiting conditions.

A scheduler contains several worker threads which execute protocol instances. It is possible to use multiple schedulers, but each protocol must be assigned to exactly one scheduler. The total number of threads used by the schedulers should be in the order of the CPU cores because protocols do not perform any blocking I/O calls. Thus, the threads are most of the time in the state ready or active and can therefore utilize the multi-core CPU sufficiently. A special GUI scheduler executes protocols in the GUI thread which allows these protocols to provide a graphical user interface.

Protocols communicate asynchronously by exchanging messages as shown in Figure 1. Thus, they operate highly decoupled. This is useful when the application must scale up to multiple computers. As long as protocols communicate by message exchange only, they can easily be distributed. Each received message is queued in the broadcast message queue.

Every yield-return-statement returns a waiting condition. It is possible to wait for a message, a timeout, a signal, or the completion of another state machine. The protocol's scheduler is constantly watching whether any of the waiting-conditions is fulfilled. In this case the corresponding state machine (i.e. iterator) performs a state transition and yields a new waiting condition.

The concurrent state machines inside a protocol are executed pseudo-parallel by cooperative multi tasking. At every yield-return-statement control (i.e. the worker thread) can be passed to another state machine. A design rule of Gears4Net is that the code between two yield-return-statements must

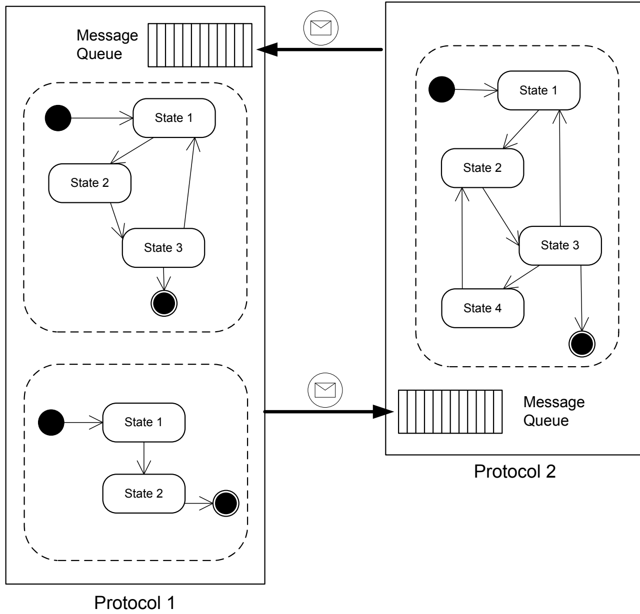

Figure 1. A Typical Gears4Net Application

transfer a consistent state into another consistent state. However, only one worker thread at a time is allowed to execute inside a protocol. This tremendously simplifies the implementation since inside a protocol there is no need to lock any data structures. In fact, typical Gears4Net applications have no lock at all: inside a protocol it is not needed and between protocols there is only an asynchronous message exchange. The pseudo-parallel state machines inside a protocol are very convenient for implementing application-layer network protocols, which feature inherent parallelism.

As mentioned earlier each protocol features a broadcast message queue. Whenever a yield-return is encountered, the protocol tries to dequeue a message from the broadcast queue. The message is delivered to all state machines which are waiting for this message. In case nobody is waiting for the message, the message is dropped. The reason why we call it broadcast is that a) one message can be received by any number of state machines and b) the message is broadcasted even if nobody is listening. This is a major difference to traditional queuing disciplines where queued messages must be explicitly dequeued by someone who waits for the message.

Our experience with Gears4Net has shown that the traditional queuing can easily lead to deadlocks. Imagine, someone is sending a message to a protocol but currently no state machine wants to receive it. This unwanted message would plug up the message queue and result in a deadlock. The only possible relief would be to drop the FIFO semantics of the queue which would lead to unwanted out-of-order execution. Thus, by default the broadcast queue is used.

Nevertheless, some traditional queuing examples are not possible with our broadcasting queue. For example, the famous producer/consumer example. The reason is that the producer will never block because sending is asynchronous. If a fast producer is sending too many messages, a slow consumer may have to drop some messages or otherwise the queue length will grow forever. For these cases, Gears4Net features traditional queues, too.

Gears4Net encapsulates the described protocol features in an abstract class named ProtocolBase. Hence implementing a custom protocol begins with the creation of a class inheriting from ProtocolBase. Listing 3 shows the MyProtocol class implementing the Execute-iterator defined in the abstract ProtocolBase class. This iterator is the main state-machine of the protocol. Once the protocol is started the execute-iterator will be invoked. The protocol will be executed until the main state-machine terminates. In case of the given example the main state-machine will terminate immediately if no programming logic will be inserted in Line 3.

```
1   public class MyProtocol : ProtocolBase {
2      public IEnumerator<ReceiverBase> Execute() {
3         //1st state machine - insert programming logic
4         yield break;
5      }
6   }
```

Listing 3. Implementing ProtocolBase

Listing 4 demonstrates a typical network application scenario. We are sending a request message to an entity and are waiting for a response. Therefore the BroadcastMessage-method is invoked and a request-message is passed as a parameter. The method will return immediately and the yield return statement including the receiver waiting for the response-message in Line 3 will be executed. Once the response-message is received the iterator will continue its work. The plus operator in Line 3 indicates that the operator on the right side will be executed after the operation on the left side is completed. Hence, the HandleResponseMsg-delegate (type safe function pointer) will be invoked when the receiver receives a response-message. In general, a handler can either be a normal C# function of the protocol class or another state machine (i.e. iterator) that is launched when the message is received.

```
1   public IEnumerator<ReceiverBase> Iterator()  {
2      BroadcastMessage(RequestMsg);
3      yield return Receive<ResponseMsg>() +
          HandleResponseMsg;
4   }
```

Listing 4. Receiver and broadcast message queues

The semantics of the plus operator requires that the right side of the operator can access the result of the left side. Therefore, Gears4Net evaluates if the return value on the left side can be passed directly to the delegate of the right side. The type checking mechanism will be executed implicitly by the C# compiler. A type missmatch results directly in a

compile-time-error.

Listing 5 extends the described network application by a timeout scenario. We are sending a request message to an entity and we are waiting for the response as in the previous example. But additionally we would like to set up a timeout-condition. If the response-message is not occurring within a defined timeframe we would like to trigger a timeout event.

The scenario proposes that we are waiting either for the response-message or the timeout. Thus we have to extend the expressiveness of the wait-conditions by introducing the or-operator. The semantics of the notion of `a|b` is the following. If `b` occurs first, execute the right side of the operator. If `a` occurs afterwards ignore `a` . And of course vice versa.

Listing 5 uses a combination of timeouts and the or-operator to implement the discussed scenario. The usage of the or-operator as well as the timeout-statement can be found in line 3. Thus, the timeout triggers when the response-message does not arrive in five seconds.

```
1   public IEnumerator<ReceiverBase> Iterator()  {
2      BroadcastMessage(RequestMsg);
3      yield return Receive<ResponseMsg>() +
           HandleResponseMsg | Timeout(5000);
4   }
```

Listing 5. OR-operator and Timeout-statement

We have discussed the handling of asynchronous message broadcasting within a protocol. This mechanism is extremely powerful to distribute content-data into the state-machines but oversized to simply indicate that an event has taken place. Hence, we are introducing the signal-concept to down-size the overhead of sending body-less messages. Listing 6 presents an example using a signal. The signal is created as a class member in Line 1. Receiving a signal requires the call of the CreateReceiver-method on the signal itself as shown in Line 3. A signal triggers an event when the Emit-method of the signal is called. Emit may be invoked from any thread in any protocol. Thus, we implemented a very lightweight solution to signal events.

```
1   private Signal mySignal = new Signal();
2   public IEnumerator<ReceiverBase> Iterator() {
3      yield return mySignal.CreateReceiver() +
           executeHandler;
4   }
```

Listing 6. Signals

The signal concept is very useful in various scenarios. Signals are, for example, the common solution to terminate a state-machine as shown in Listing 7. Furthermore, we want to handle up to four messages in parallel. Thus, we are presenting the Parallel-command as shown in Listing 7. The parallel-command can be invoked with different parameters. Line 2 shows its most frequently used parameter list. The maximum number of messages processed concurrently can be limited by the first parameter. The example handles a maximum of 4 at a time. The second parameter of the parallel-command is the receiver that indicates what we are waiting for. The third parameter is the soft-termination-condition. When the `softExitSignal` triggers, the parallel-command stops handling further messages, but already running handlers are allowed to complete. This supports a clean shutdown of message processing.

```
1   public IEnumerator<ReceiverBase> Iterator()  {
2      yield return Parallel(4, Receive<Msg>() +
           HandleMsg, softExitSignal.CreateReceiver());
3   }
```

Listing 7. Parallel execution

By default Gears4Net receivers are typed, i.e. in the brackets you can describe the type of message you are waiting for. Sometimes further filters are required, which inspect the content of the message and not only the type. To achieve an easy to use filtering mechanism we are using the functional-programming concept of lambda-expressions which is even found in object-oriented languages such as C# and Python. Listing 8 shows a lambda-expression embedded into the receive-command. The expression `m => m.Source == "a"` evaluates to true if the message source equals `"a"`.

```
1   Receive<Msg> (m => m.Source == "a") +
        HandleResponseMsg;
```

Listing 8. Filtering messages using lambda expressions

With Gears4Net it is possible to launch one or multiple state machines and wait until one or all have completed their work. Listing 9 presents the Launch-command that instantiates new state-machines. The wait-condition tells Gears4Net to launch the state machines `Iterator1` and `Iterator2` and wait until at least one state-machine terminates. If `Iterator2` terminates first, the wait-condition will be fulfilled and `Iterator1` will be terminated implicitly. For these semantics, we used the |-operator. In contrast, we could write `Launch(Iterator1) & Launch (Iterator2)` to indicate that we want to wait until both state machines have completed their work.

```
1   public IEnumerator <ReceiverBase> Iterator () {
2      yield return Launch (Iterator1) | Launch (
           Iterator2);
3   }
```

Listing 9. Waiting for the state-machine terminating first

The examples so far have launched new state-machines within a yield-return statement. The intention there is to create the state-machine and wait for its termination. Launching a new state-machine at an arbitrary point inside the protocol and waiting somewhere else for its termination is impossible with the concepts we have shown so far. Thus, we are introducing the concepts of LaunchDetached and Join.

Listing 10 shows how to start two state-machines using the LaunchDetached-command which creates a new `AbstractStateMachine` from some iterator. Anywhere inside the protocol one can use the Join-command to wait for the completion of any detached state-machine. In the

example the yield-return-statement returns a waiting condition which waits for both state-machines `sm1` and `sm2` to terminate.

```
1   public IEnumerator <ReceiverBase> Iterator () {
2       AbstractStateMachine sm1 = LaunchDetached (
            Iterator1);
3       AbstractStateMachine sm2 = LaunchDetached (
            Iterator2);
4       // execute custom code
5       yield return Join (sm1, sm2);
6   }
```

Listing 10. Starting state-machines detached

The previous examples discussed several aspects of the Gears4Net framework. In the following we discuss a more complex scenario. The diagram in Figure 2 shows a common network-based scenario. Starting at the entry point of the diagram we are trying to establish a network connection to a remote host. The entire protocol (connection establishment plus processing) must finish in a certain time which is guaranteed by the timer on the right side. Upon connection establishment, the state machine enters three parallel states. The Connected state waits until the connection is closed. The Timer state at the right side of the diagram checks that the processing itself does not take forever. The application logic is implemented by the Application sub-state-machine. If one of these three states is left, the entire protocol will change to the Closing state and stops. If for example the connection is closed before the Application sub-state-machine is finished, the sub-state-machine will be terminated implicitly. There are several ways to handle the forceful termination of a state-machine which are not discussed in this paper.

The example shows that even complex parallel constructs can be implemented in Gears4Net with little effort (see Listing 11). Furthermore, race conditions that could occur due to messages and timeouts arriving at the same time in different threads are avoided. In addition, it is not possible to forget the reset of some timer. Our experience has shown that Gears4Net code is less susceptible to race conditions.

```
1   public IEnumerator <ReceiverBase> Iterator () {
2       OpenConnectionAsync ();
3       yield return ( Receive<ConnectedMsg> () +
            StartApplication | Timeout (5000) | Receive<
            ConnectionClosedMsg> ())
4         | Timeout(10000);
5       CloseConnection ();
6   }
```

Listing 11. Gears4Net implementation

It should be clear by now how Gears4Net can be used to realize applications which have a high degree of inherent parallelism. Sometimes Gears4Net even reduces parallelism, e.g. by allowing only one worker thread in each protocol, although there are concurrent state-machines inside a protocol. The rational of our design is that there is no gain in writing programs that could utilize 20.000 CPU cores if the current state of the art offers core numbers below 100. This extra parallelism will therefore not contribute to speed-ups,

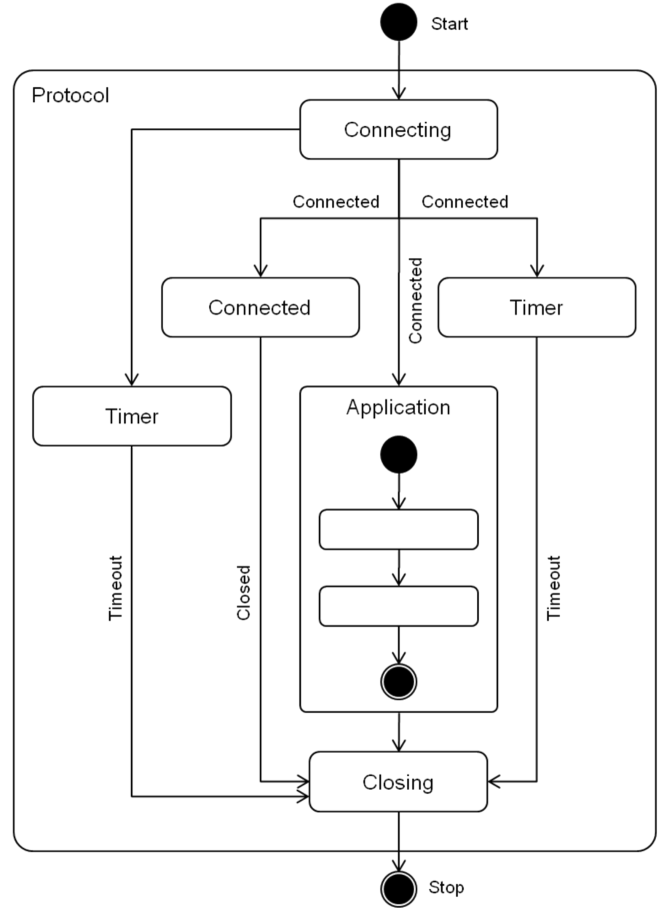

Figure 2. Network application state-machine example

but it allows for nasty race conditions. Protocol instances are really running in parallel in Gears4Net. To avoid race conditions, dead-locks etc. we have chosen asynchronous message passing as the only communication scheme. This design decision allows us to scale out by distributing protocol instances across several servers.

V. Related Work

Since the rise of multi-core CPUs, several programming models have been suggested to improve parallel programming. While some of this work inspired Gears4Net, their aim is different. These programming models foster parallelism in applications which are not inherently parallel. The reason is to speed up these applications by utilizing more than one CPU core at a time.

The aim of Gears4Net is different. When handling thousands of concurrent TCP connections or simulating thousands of peers, parallelism is inherent to the application domain. Gears4Net allows us uphold parallelism while using a limited number of threads and reducing the memory overhead. Nevertheless, we discuss some of the other programming models here since they offer synchronization

paradigms which inspired Gears4Net.

A. *Comega*

The first one is Polyphonic C# which is part of a language extension of C# called Comega [3]. The basic concepts of Polyphonic C# are asynchronous method calls and chords (join pattern). Commonly a method call is a blocking operation that returns when the method body was executed. In contrast to synchronous method calls, Polyphonic C# defines a new keyword, async, so that method calls return immediately while the body of the method is concurrently executed e.g. in another thread. Chords are the second concept in Polyphonic C#. A chord contains one or more asynchronous methods and a body. The body will be executed when all asynchronous methods have been called. In comparison, Gears4Net allows to start state machines (i.e. an iterator function) asynchronously (via LaunchDetached) and wait for their completion (Join). However, in Comega asynchronous functions are executed in parallel, i.e. in different threads to utilize multi-core CPUs. In Gears4Net the state machines of one protocol are executed pseudo-parallel, i.e. at most one is executing at a time. This reduces the need for synchronization inside a protocol. CPU utilization is given because a typical Gears4Net application runs many protocol instances which can be executed in parallel, i.e. in different worker threads.

B. *CCR*

The second parallel programming model we discuss is a library called Concurrency and Coordination Runtime (CCR) [4]. CCR supports port based programming, which is inspired by the Pi-calculus. A thread can send messages to a port, which queues them up. Furthermore, it is possible to wait for a message arriving on some port and handling it in some worker thread. CCR was to the best of our knowledge the first such framework to utilize the C# iterators in the same way Gears4Net does. Thus, the design of Gears4Net builds on the concepts of the CCR. However, CCR does not feature broadcast-message queues which are essential to the design of Gears4Net. Furthermore, CCR features schedulers and state machines executed by the schedulers. The concept of a protocol executing state machines in pseudo-parallel mode is not present in CCR and cannot be implemented on top of it. Some synchronization constructs such as Launch/Join and Signals are missing in CCR, too.

C. *Path Expressions*

The waiting conditions of Gears4Net are a super-set of path expressions [5]. Path expressions are a comparably old concept for describing permitted sequences of execution. For example, "`{4:read},write`" specifies that up to 4 reads in parallel are allowed or one write. In Gears4Net this can be written as

```
yield return Parallel(4, 4, read)^ write
```

Thus, path expressions have been incorporated in the design of Gears4Net, but the concepts of Gears4Net such as signals, queues, launch/detach go far beyond the original path expressions.

Campbell and Kolstad showed in [6] how to integrate path expressions in the programming language Pascal. Gears4Net avoids modifying the programming language due to the usage of the C# iterator concepts. Furthermore, the Pascal path expressions are handled by the compiler, i.e. they are static. In Gears4Net the path expressions are constructed and evaluated at runtime which allows for higher flexibility. Especially when waiting for complex events the nature of the event is only known at run-time and compile-time path expressions are of little help.

D. *Monitors*

Hoare [7] and Brinch Hansen [8] developed the concept of monitors. The basic idea of monitors is the monitor condition: at any time at most one thread is allowed to execute inside the monitor. In Gears4Net each protocol fulfills the monitor condition, i.e. at most one worker thread is allowed to drive a state machine inside the protocol. The difference is the way of invoking methods in a monitor and in a protocol. In traditional monitors a thread invokes a function inside the monitor. This thread is put to rest until it can enter the monitor. In Gears4Net protocols only communicate by exchanging messages. No thread will ever be put to sleep.

E. *Seda*

Welsh introduced the staged event-driven architecture (SEDA) [9] and proposed tasks, thread pools and queues as the core building blocks for SEDA applications. The SEDA design envisages a series of independent stages separated by queues. Each stage processes a task dequeued from its incoming queue and pushes it into the incoming queue to the next stage. The task is executed by thread-pool thread and laid back afterwards. The Gears4Net concept uses a related actor pattern but differs in thread-scheduling and queuing patterns as well as in the state-machine like programming model.

VI. Conclusion and Future Work

Gears4Net is a framework for implementing highly parallel networking applications. The main goal of our work is to provide developers with an easy to understand programming model that minimizes resource usage. We have argued that threads provide no reasonable programming abstraction because a) current main stream operating systems are not designed for thousands of threads and b) the stacks for thousands of threads allocate too much memory anyway. Furthermore, we pointed out that complex event detection is a difficult topic, especially in highly parallel applications.

Gears4Net provides a programming abstraction based on C# iterators and non-blocking I/O. This approach features a near minimal memory footprint and uses only a constant number of threads. The developer can design his software as a set of concurrent state machines. Tightly-coupled state machines are grouped inside protocols, are automatically synchronized and have access to common state. State machines in different protocols communicate by asynchronous message passing only.

We have several man-years of experience with Gears4Net yet. We used it to implement and test several of our peer-to-peer systems, including peers@play [10], Symstry [2], and several peer-to-peer bootstrapping protocols [11]. Furthermore, we are developing a special web-application server on top of Gears4Net which is the basis for a Web 2.0 university spin-off (www.picoscribe.com). Some of the software has been rewritten to use Gears4Net and we experienced that the number of race conditions in the code reduced dramatically.

Approaches such as CCR and Comega lend themselves to a clear and minimal design. In contrast, Gears4Net provides a large toolbox of features, ranging from different queues, over signals, timers, and joins, to concepts similar to path expressions. We believe that developers rather appreciate one additional concept than to study literature on how to solve their problem with a minimal set of concepts.

In the future we will investigate the problem of complex event detection more closely. Using Gears4Net it became very easy to write complex event conditions, i.e. waiting for a combination of messages, joins, signals, timeouts etc. Developers started implementing features that would have been very complicated and error prone without Gears4Net. As a result the waiting-conditions became larger and larger. We will investigate how to improve the internal design and speed of the waiting-condition.

REFERENCES

[1] University of Duisburg-Essen and University Mannheim, "peers@play homepage," published on the WWW at http://www.peers-at-play.org/, 2008.

[2] M. Saternus, M. Knoll, F. Dürr, and T. Weis, "Symstry: Ein P2P-System für Ortsbezogene Anwendungen," in *Proceedings of 15. ITG/GI - Fachtagung (KiVS 2007)*. VDE-Verlag, Februar 2007, Konferenz-Beitrag, pp. 99–104.

[3] N. Benton, L. Cardelli, and C. Fournet, "Modern concurrency abstractions for c#," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 5, pp. 769–804, September 2004. [Online]. Available: http://dx.doi.org/10.1145/1018203.1018205

[4] S. Singh and G. Chrysanthakopoulos, "An asynchronous messaging library for c#," *Synchronization and Concurrency in Object-Oriented-Languages (SCOOL)*, at OOPSLA October 2005.

[5] R. H. Campbell and A. N. Habermann, *The specification of process synchronization by path expressions*. London, UK: Springer-Verlag, 1974, pp. 89–102.

[6] R. H. Campbell and R. B. Kolstad, "Path expressions in pascal," in *ICSE '79: Proceedings of the 4th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1979, pp. 212–219.

[7] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, pp. 549–557, 1974.

[8] P. B. Hansen, "The programming language concurrent pascal," *IEEE Trans. Software Eng.*, vol. 1, no. 2, pp. 199–207, 1975.

[9] M. Welsh, D. E. Culler, and E. A. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. Banff, Alberta, Canada: ACM Press, October 2001, pp. 230–243.

[10] A. Wacker, G. Schiele, S. Holzapfel, and T. Weis, "A nat traversal mechanism for peer-to-peer networks," in *P2P '08: Proceedings of the Eighth IEEE International Conference on Peer-to-Peer Computing (P2P'08)*. Aachen, Germany: IEEE, Sep. 2008, pp. 81–83.

[11] M. Knoll, A. Wacker, G. Schiele, and T. Weis, "Decentralized bootstrapping in pervasive applications," in *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications*, 2007.