Self-Tuning Planned Actions
Time to Make Real-Time SOAP Real

Johannes Helander
Microsoft Research

Stefan Sigurdsson
University of Washington

Abstract

This paper proposes a new method for programming and controlling distributed tasks. Applications declare behavior patterns that are used to automatically predict and reserve resources needed by applications in a heterogeneous distributed environment. The paper introduces a stochastic quality sampling driven scheduler and a rendezvous mechanism for matching pre-planned activities with actual payload data.

The system is built around the first real-time SOAP implementation, also presented in this paper. It extends the XML Web Services interoperability benefits that have proven themselves in e-business into two new areas: embedded and real-time. The paper presents an efficient implementation that runs on common microcontrollers and other computers.

1. Introduction

Embedded real-time computing systems for entertainment and other consumer–as well as industrial–uses are increasingly built out of distributed components that are manufactured by multiple vendors. This means that interoperation becomes the number one issue in any communications paradigm. Meeting real-time predictability standards becomes ever harder as the devices get more diverse and the number of temporal uncertainties in their interaction increases.

This paper proposes a self-tuning planning mechanism for such environments. It uses an on-line, sampling driven statistical model to predict resource requirements needed to meet application's quality of service needs. Open standards based XML Web Services are used both for application and for system communication in order to maximize interoperability.

The system attempts to keep applications simple by separating temporal behavior from actual work. The separation allows an automated advance planning phase and facilitates power savings and real-time guarantees through admission control.

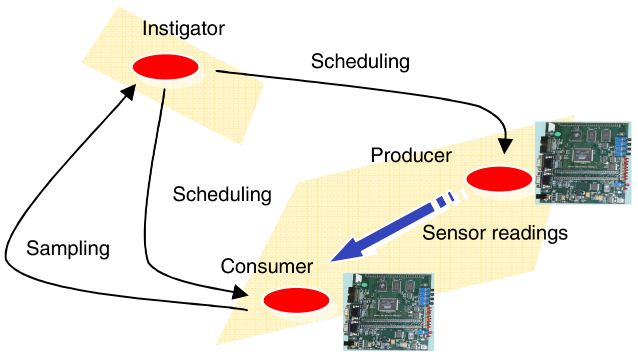
Temporal behavior is represented by *behavior patterns*. The patterns describe the resources needed at each node involved in a task and when they are needed. The description is expressed in a declarative high-level language in XML syntax. Patterns are instantiated into *planned actions* that are negotiated with the schedulers at each node.

While the scheduler is tuned to work with incomplete information, it carefully preserves the knowledge that is available by extending a precise constraint based scheme (time constraints are defined in [5]). These extended constraints, coined *planned actions*, can exist before the precise execution parameters have been determined. This allows a time delay between separate planning and execution phases. Through inference from the predicted execution it is possible to determine when resources are not needed and can be shut off as well as for clustering usage. The longer a device can be turned off the more power is saved.

The scheduling is divided into distinct phases, both temporally into long and short term plans and spatially into distributed task and local node processing. Execution and control flows are separated and a continuation based rendezvous mechanism combines them back together. A wild card rendezvous captures both time driven spontaneous execution and message driven execution [4] into a common abstraction. An implementation that is available to the public and runs on multiple platforms is introduced and measurements show that it is functional and highly efficient.

1.1. Sample Applications

Examples of usage areas include: 1) A home entertainment and automation system built by multiple manufacturers. Different media types have different jitter tolerances, some activities are repeating and others are not.



Various inhabitants instigate multiple activities. Energy efficiency reduces noise and unwanted heat. 2) Industrial robot control. An assembly line stops for 10 seconds every minute. Welding a door handle takes 4 seconds. If the robot welds a door handle outside the correct time window, it destroys the car. A proportional schedule would not work. 3) Sensor and actuator data streaming. The figure above examines #3 in further detail and the performance section shows measurements.

A sensor node reads a sensor at regular intervals and sends the data to an actuator node that reflects the sensor state. The actuator node knows when to expect data from the sensor so it can turn off its receiver while *not* expecting data. Energy efficiency lengthens battery life. An instigator node initiates the data flow and adjusts the timing according to measurements (quality control sampling). There can be multiple instigators as well as multiple sensor nodes.

1.2. Key Contributions

The system described in this paper:

Extends stochastic quality control paradigm to distributed embedded computer scheduling and allows adaptive real-time scheduling in the presence of many unknowns such as is the case in heterogeneous systems.

Splits distributed jobs into behavior pattern, temporal instantiation, spatial instantiation, service objects, and data driven actual work. This allows an advance planning phase facilitating a simple but informed and power-efficient low-level scheduling during the execution phase. The resulting system automates programming tasks allowing more abstract development.

Applies XML Web Services to a completely new area and lets real-time applications interoperate with legacy systems in a meaningful way.

2. Architecture

Distributed tasks communicate through messages and execute service methods on individual nodes. The middleware described in this paper exploits similarity in communication patterns and resource needs between similar tasks that exhibit *behavioral patterns.* The system separates the temporal and spatial behavior and controls execution and scheduling based on the patterns and predicted behavior and needs. The planning is done at an earlier time than actual messaging and communication thus facilitating resource planning at an earlier point of time than when the execution actually occurs. The actual data later drives the actual work but the plan pre-reserves the necessary resources assuring that the actual work can be completed successfully. Due to variations to the actual work and random delays, the planner needs to over-reserve resources to account for the variations. A statistical model is used to predict just exactly how much needs to be reserved to assure the level of quality a specific task needs. The more critical the application, the higher the required quality is and the more resources it needs.

An *instigator* is the application that drives a task. The instigator declares a behavior pattern and the criticality of a task as well as medium dependent tolerance to jitter and the desired deadline when the specific instance of the task should be completed. The middleware planner instantiates the behavioral pattern into a spatial and temporal plan. The middleware derives a specific action and resource plan for each node that is needed to execute the task (that is an instance of the pattern). It then negotiates the resources with each node. The application then gets the task going by calling a method on an initial object that makes the middleware send an initial message to an initial service.

While the task executes, the middleware monitors its progress and measures its timing and resource consumption. The middleware on the worker nodes send samples of the collected information back to the planner on the instigator according to a predetermined sampling schedule. The instigator planner adjusts its model that produces updated predictions on the resource needs. If the quality of the measured performance does not match that specified by the application, the planner will attempt to renegotiate the plan with the worker nodes. Commonly the adjustment will be downwards as the quality of the stochastic distribution itself increases with more samples and thus less over-reservation is needed.

The rest of this section examines in further detail behavior patterns, their instantiation through planning, how sampling is used to derive improved plans and adapt to changes, the scheduling and mechanisms needed at each node, and how the actual execution is matched with the given plans.

2.1. Behavior Patterns

A behavior pattern identifies what sequence of actions is needed, what messages are expected, and what types of resources are required to execute a task. The behavior does not express where, when, and how much, neither does it express the content of the messages or the method for processing the messages. It does name the object that provides the methods, however.

It is envisioned that behavior plans can eventually be automatically created, either based on a model or by observation of instrumented execution. The patterns used for this paper have been hand authored. However, their temporal instantiation is automated by the planner as will be shown.

```
<behavior name="SensorDemo">
  <action name="DemoInstigator" endpoint="node:instigator/COB/sensormain.cob">
    <message destination="SensorProducer/*"/>
  </action>
  <action name="SensorProducer" endpoint="node:sensor/COB/sensor.cob"/>
     <repeat count="100" Period="P1.5S"/>
     <message destination="SensorConsumer"/>
   </action>
   <action name="SensorConsumer" endpoint="node:consumer/COB/sensor.cob"/>
     <repeat count="100" period=”P1.5S”/>
    </action>
   <sampling destination="node:instigator" interval="20" number="2"/>
</behavior>
```

FIGURE 2: Example behavior pattern for three node demo scenario.

The pattern in figure 2 expresses a pattern for the sensor demo [figure 1] that runs on three nodes. The instigator sends an asynchronous message to *sensor* that is directed to all 100 of the SensorProducer action instances. The sensor producer then runs the method expressed in the eventual message for one hundred times, each run offset by a second and a half. The producer also sends a message each time to the consumer, which expects one message per action instance. The pattern also states that sampling should be done at a double sampling schedule every 20 invocations.

The pattern is instantiated to spatial plan by a discovery service. It resolves the roles into precise URLs and network addresses. The discovery, trust, and security issues are discussed in a companion paper [3].

The pattern is instantiated temporally by the planner. The planner uses a stochastic process and sampling to predict how many resources are needed for a given application specified quality standard. It estimates how much time will be needed at each node and how far before a deadline an action needs to start. The planner also offsets the various actions relative to each other so that the overall deadline can be met.

The specific start time is finally calculated from an application supplied overall deadline. The fully instantiated pattern is now an action plan, a list of actions with their corresponding locations, resources, and times. After this the middleware negotiates the resources with the worker nodes. It is now up to the application to provide the data for the initial message that sets the plan in motion.

2.2. Planned Actions

Planned actions express temporal behavior of a program. Action scheduling is an extension of earliest deadline first and constraint based scheduling. Planned actions are also instantiations of behavior patterns, bound to a specific time and place. The actions list what resources are needed for a task, at what time those resources are needed, and where those resources are located plus how much time variance can be tolerated. At any one time and place the action is a multi-dimensional resource vector that enumerates all the various resources needed (CPU cycles, memory, I/O bandwidth, etc.). The planned actions also describe the relationship between related actions through triggers. One action can trigger another action or itself.

```
<task name="SensorDemo-123456">
  <action name="SensorProducer" deadline=”2004-12-11T02:51:48.7001508Z”
           tolerance=”P0.005S” duration=”P0.02S”>
     <trigger maxCount="100" offset="P1.5S">SensorProducer</trigger>
   </action>
   <sampling destination="http://10.10.10.10/COB/statistics.cob" interval="20" number="2"/>
</task>
```

FIGURE 3: Action plan for producer node that corresponds to the pattern in figure 2.

The planned action model proposed in this paper is an extension of the constraint based scheduler (CBS), itself and extension of earliest deadline first (EDF). Rather than having a thread declare its constraint, the resource plan is separated from execution. The action plan is a constraint with a state machine, where execution is possible only in the right state. Once an action is entered into the local schedule, it reserves the time just like a constraint. However, unlike a constraint, a planned action needs more than just the correct time to run. First it needs to be triggered (see below) and secondly it needs to have something (a continuation) to run associated with it. In other words, *what* is executed is separated from *when* something is executed and in *which sequence*. The action's triggers control the sequencing, the action's deadline control when. The *what* is controlled by the implementation of the services and the methods that are called and the specific patterns – in other words the content of the payload messages.

A planned action item is a tuple <Deadline, Estimate, Tolerance, Triggers; SeqNo, State; → Consumed>. The deadline is the same as in EDF and CBS, the Estimate is the same as in CBS, The Tolerance corresponds to the Start time in CBS (Tolerance = Deadline - Estimate - Start) but is a somewhat more meaningful number to applications. The Triggers is a list of actions to trigger once the current action is completed. The sequence number distinguishes one instance of a repetitive action from another. Once a worker node has accepted an action, the middleware executes a state machine on the action. The state is one of initial, wait-trigger, wait-message, wait-start-time, run, terminated. The service application gets to execute once the state machine reaches the Run state.

S (*msg*)→WTrig (*trig*)→ WTime (*now>Start*)→Run→T
(*trig*)→ WMsg (*msg*) ↗

A repetitive action is one that triggers itself with a time offset. A sequence of actions is one where actions trigger each other in a chained fashion. A one-time action is one where there is only one action that does not trigger anything. An action that is not triggered by any action other than itself is considered automatically triggered initially. A trigger is the tuple <WhatAction, TimeOffset, RepeatCount>

While actions form trees in the middleware that creates and interprets them, they are represented as simple XML blocks while in transit. The messaging middleware automatically serializes them. An example of a serialized planned action is shown in figure 3.

2.3. Action Scheduling Roles

Scheduling of planned actions is split into three distinct phases: 1) instantiating a behavior pattern into a concrete action plan by the instigator (planning); 2) negotiation of the plan with the worker nodes (admission control); and 3) executing the action plan and state machines on the worker nodes.

The planning (#1) is done by an instigator, who drives a specific task. There may be multiple instigators driving multiple instances of multiple tasks. Each task can be planned in isolation and its resource needs can be estimated in isolation. Similarly each node knows how to best execute the work that is delivered to it (#3). It will, within the given constraints, try to execute the schedule in an optimal order, presumably to save energy.

The admission control (#2) can either be done at the worker node or it could be done at a central machine that knows the schedules for all the machines. The advantage of the decentralized model is that it works in an ad-hoc peer-to-peer type environment. The advantage of the centralized model is that the central node could be more powerful and have more complete information. A central node could also schedule shared resources such as radio bandwidth and pick the best node to execute a given action when there is a choice. The implementation described in this paper provides a simple decentralized admission checker.

It is interesting to note the temporal shift in the scheduling in addition to the location differences. The planning and admission control are done at the time of instigation, when work is initially started (e.g. the TV remote was pressed to start a movie). The execution of the plan, the local scheduling, is done much later, while the movie is playing, with full knowledge of everything that needs to be done. This allows the local scheduler flexibility as it knows all it needs to do. It can for example attempt to cluster all execution so as to turn off itself for the time in between. Note that an action plan for listening for button presses on the remote is needed for the most aggressive power savings not to shut out task creation. Similarly plans can be provided for other spontaneous activities with the expected patterns that follow.

An action plan does not mandate that any execution actually results. It is merely a reservation that guarantees that the resources will be available when needed. The cost of a task thus consists of a reservation cost in addition to the work itself. The reservation cost depends on what other plans were not admitted as a result and depends on the amount of resource in addition to the degree of inflexibility. In other words a plan that requires high confidence and little jitter is more expensive as there is more potential conflict with other plans.

2.4. Estimating Resource Consumption of Actions

The behavior pattern given to the scheduler lists some of the resources that are needed for executing instances of the pattern. Others, in particular CPU and memory, are needed by almost all actions so those can be inferred automatically. Yet more can be detected in the sampling phase. The planner also needs to estimate overall time. The simplest planner only needs to consider overall time. This is the case with the planner presented in the implementation section of this paper. A multi-resource scheduler that also deals with real-time C# has been demoed by the author at the Microsoft Faculty Summit and will be presented in a separate paper.

The heart of the planner is a stochastic filter. It stores previous samples in a table. It calculates the distribution of the samples. It integrates the distribution up to the application supplied confidence interval (see figure 4). The integral gives the amount of resource (time) required to assure the work will be complete within the interval with the given probability (the confidence). The size if the integral depends on 1) the median 2) the deviation (volatility) of the measurement, and 3) the quality of the measurement itself. In order to stabilize the estimate, a negative feedback loop is added. This works by mixing the old median and deviation into the new estimate when new samples replace the old.

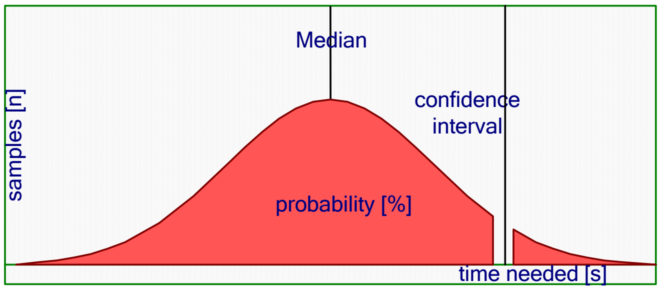


FIGURE 4: The confidence is the probability that a task completes in the time reserved. The reservation needed is the inverse integral of the distribution up to the required confidence.

The planner uses the samples to calculate a distribution function. It reserves enough resources to achieve the required quality (confidence). The higher the required quality the more resources are needed. Similarly, the more jitter there is in the measurements the more resources will be reserved. Uncertainty about the distribution itself also adds to the resource requirements. Initially, when there are no measurements, the planner uses application supplied guesses. A dry-run can also be used to prime the distribution function.

The implementation uses pre-calculated values of the integral of the normal distribution and fixed point integer arithmetic to avoid expensive calculations. The integral is roughly log10 of the confidence, e.g. when going from 99% probability to 99.99% probability the amount of time needed increases by twice the median.

If the planning is done in a centralized way, or as a service in its own right on a powerful machine, the calculations could be done by e.g. Matlab or some modeling tool but a microcontroller needs to keep the calculation to a minimum, while pre-calculated values can be a significant saving.

One of the benefits of the stochastic approach is that it works even when information is incomplete. When the quality of information increases the planner is able to make more tight plans but even in the presence of significant uncertainty (like non-real-time nodes) it can still make a plan with confidence.

2.5. Quality Control Sampling

The sampler measures the time and resources consumed at each invocation on the worker nodes with the help of the operating system. The middleware sends samples back to the planner according to a predetermined sampling schedule. By adhering to the schedule, the sampler avoids overwhelming the network and the nodes themselves with excessive sampling messages and keeps the overhead at an acceptable level while still facilitating a sufficient quality control and accurate predictions.

The planner integrates the new samples into the distribution function. When it detects a significant enough change, it renegotiates the action plans with the worker nodes and either releases resources for other use or requests more as appropriate.

2.6. Scheduling of Action Plans

Once the planner has instantiated the location and timing of the behavior pattern, it negotiates with each required worker node for acquiring a reservation for executing the plan. It sends a task description [Figure 3] to the worker node using SOAP. The scheduler on each node performs an admission check and checks for conflicts. If there are no conflicts, the task is accepted. Otherwise the task is rejected and the instigator application is notified of failure. If any of the worker nodes reject its plan, the planner cancels the task on all the worker nodes. At this point the application can take corrective action and attempt to reschedule at another time. If instead the plan is accepted on all nodes, control is returned to the instigator application for making the initial method call that results in the first message being sent that in turn puts the plan into action.

Note that the middleware can piggyback the initial method message with the initial reservation request. If the reservation fails, the method message is ignored. The implementation employs this piggybacking to save one roundtrip. The success of the method message implies the acceptance of the task.

Each node keeps a list of all accepted tasks. When a new task is proposed or an old one is adjusted, the local scheduler runs an admission check to determine whether there are conflicts. This is essentially a binpack problem with many known solutions [Knuth, MP-scheduling, nurses] but is also NP-complete, meaning that a precise answer is not feasible. Thus a best effort algorithm that sometimes produces false positives is used.

The conflict detector first attempts to determine conflicts against any repetitive actions and then attempts to fill in one-time tasks into the gaps. The algorithm uses a calculated slack that is initially set to the jitter tolerance given by the application. It then compares the new task against one other task at a time. When a potential overlap is detected (the period and time estimate interfere at any point in time) the calculated slack between the two tasks is adjusted by reducing some slack from each of the actions so that the sum of the adjustment equals the overlap. If at any point the calculated slack becomes negative, the new task is rejected and the calculated slack of the existing nodes is returned to what it was before the insertion.

In essence the admission check is a tree merge of two scheduling trees, if the merge fails the admission is rejected. The tree merge is location independent as long as there is a single authority. Thus in a centralized scheduling system the authoritative scheduler will simply deliver the current tree to the worker node.

The worker node uses the scheduling action tree to drive the timing of actual execution. A low-level scheduler handles context switching, priority inheritance and similar short-term issues.

One big advantage of having pre-declared long-term scheduling plans is that the scheduler can know when execution is *not expected*. The scheduler can use this information to shut off the power during those times. In addition the scheduler can attempt to make those idle gaps as large as possible by packing the schedule in such a way that expected execution is clustered together. Thus, when the system is powered up it can do as much as possible and then stay powered down for extended periods of time.

In the presence of multiple instigators, the admission control and negotiation mechanisms are exactly the same. The admission checker handles one request at a time.

2.7. Continuations and Messages

When a program executes in a thread, method calls are made on the thread's stack. A method call is represented by a *continuation*, which consists of a stack frame—the closure of the call arguments, the object that is being called, the method that is being called, and where to return once the call completes. In the middleware, continuations are first class objects and can exist without the thread, including its full stack. A continuation thus represents either a client side method call that is waiting for the call to complete or the server side call that is to be executed. By separating the continuation into a separate object, a continuation needs an associated thread and stack only while the continuation actually executes. This enables significant memory savings on continuations that are not currently executing.

The table driven serializer and deserializer operate on continuations. The (de)serializer uses a compact metadata representation to interpret continuation fields. They translate between the continuation and a message of a given format. In the case of SOAP, a message is represented in XML using rules specific to SOAP. There is no code specific to a given interface, except an automatically generated proxy object on the client, which provides transparent methods that simply call the generic interpreter with the method's descriptor and stack frame pointer as arguments. This runtime generated specific code is three to four words depending on the processor instruction set.

The metadata descriptor table is a compressed binary representation of the schemas of known interfaces and the messages that are part of the given interface. The descriptor can be generated at runtime from XML but is normally compiled offline so that it can be placed in ROM. The metadata can be extended at runtime (sometimes called reflection) but the metadata loaded at runtime must be placed in RAM, which is a usually a scarcer resource.

On the client side the thread that made a method call waits for its continuation to complete. On the server side, however, there does not need to be a thread a priori. Instead the system creates and recycles threads automatically when the continuation is ready to execute. The stack frame contained in the continuation can simply be copied into a thread stack and registers and it is ready to go. When the method call returns, the service thread returns to the middleware and does the necessary post-processing, including sending reply messages.

A method call on the server can be thought of as a filter that takes in one message and produces another. An asynchronous method call is simply one that does not produce a reply message. The middleware implementation recognizes this from the metadata and simply will not send a reply when not desired. On the client side asynchronous messages are similarly not waited for.

Continuations that represent asynchronous messages can be executed multiple times. The middleware simply copies the stack frame multiple times to multiple threads for execution.

2.8. Interaction between Method Messages and Planned Actions

Each method message carries with it a SOAP header that names the object the message is directed towards and an action identifier that allows the message to be matched with a particular action. This rendezvous mechanism ties the actions and messages back together after they were temporally separated.

The action id contains the name of the task, the name of the specific action, and optionally a sequence number. The sequence number identifies which specific instance of a repetitive action the message is targeted to. Missed sequence numbers can be dealt with in an application specific way.

```
<wsa:to>http://10.10.10.10/COB/sensor.cob</>
<wsa:relatedTo>SensorDemo/SensorConsumer/22</>
```

A missing sequence number is interpreted as the next one that was expected.

```
<wsa:relatedTo>SensorDemo/SensorConsumer</>
```

The sequence number can also be a wildcard. In this case the message is targeted to *all* instances of the repetitive action. The method is called over and over again with the same parameters for as many times and at the time the action specifies. The limitation is that wild carded method calls must be asynchronous as otherwise the client would receive multiple replies in an unexpected way. This limitation is enforced and a SOAP Fault message will be sent like in other failure cases.

<wsa:relatedTo>SensorDemo/SensorProducer/*</>

The repetitive single message captures the *spontaneous method* concept [4] into the same abstraction as event driven service methods. A repetitive action together with an asynchronous method message creates a spontaneous, time driven method.

3. Implementation

The middleware system presented in this paper consists is constructed of a number of software components written in C. Each component has a well defined interface that defines how it interacts with other components. All of the components can be compiled to a large number of microprocessors, microcontrollers, and VLIW signal processors. Some components, such as AES encryption, have also been implemented on FPGAs.

While the system can be run on bare metal, with the help of RTOS components, the middleware can also be run on other operating systems, such as WindowsXP. The measurements presented in the measurement section have been done on an Atmel Arm7 microcontroller. The RTOS components have been previously presented in [1]. The RTOS provides precise low-level scheduling as well as avoidance of priority inversion and starvation gaps that are needed for precise real-time operations. When the middle-ware is run on other systems, the timing can, however, be expected to be less precise.

The RTOS components include a TCP/IP network stack, a constraint based RT-scheduler, a component manager, a dynamic memory manager, synchronization (threads, conditions, mutexes), and device drivers.

The service middleware components include a tokenizer, an XML parser, SOAP serializer, discovery, addressing, key exchange and trust manager, an action scheduler and continuation manager, a stochastic planner, sampling, and encryption.

Sample applications include games, sensor and actuator services, various services that interoperate with ASP+ pages, etc.

Let's drill into some of the service middleware components in further detail:

3.1. Tokenizer and Parser

When a network driver receives data, it puts the data into a memory buffer. The buffer is handed to the network stack, which once it has determined the data is a valid packet on a supported transport; it hands off the buffer to the middleware. This happens through a new socket API that sets rules on how buffers are shared and reference counted. The middleware then deals with HTTP headers if that is the transport, or goes directly to the presentation level XML processor if the transport is UDP or other protocol that was completely handled by the network stack—after optionally decrypting the message.

The XML is processed in a push model directly from the network buffer as data is arriving and does not create any intermediate parse trees so as to minimize memory consumption. The parser delivers SAX style parsing events to the deserializer.

In the case of outgoing messages the same interfaces are used but the middleware object implementation produces messages rather than consuming them. The reuse of interfaces allows easily plugging in different transports, encodings, and encryption modules as desired. Zero-copy is used in all cases.

3.2. Serializer and Deserializer

The deserializer consumes parsing events and matches the incoming data with a metadata descriptor table that is a compact representation of all the interfaces, messages, and fields that the service understands. The metadata is generated from an XML system description. The same description also generates a reference manual, C and C++ headers, stub implementations of the service, etc.

The incoming messages are converted into continuations with native stack frames and data representation. A temporary heap is associated with each continuation for storing the argument data. When the continuation is eventually deleted, all the data gets deleted at once.

The parser understands a wide variety of data types, lists, trees of structures, in-out arguments, and language representations of multiple compilers. The outgoing messages are again processed in reverse and the same metadata is used,

3.3. Action Scheduler and Continuation Manager

When a message contains a SOAP header with an action plan, the middleware scheduler compares the new task with existing tasks. It uses the estimates, deadline, tolerance, and repetition with a slack reduction algorithm to make the determination. While the middleware implements a trust manager that can determine whether the requestor is trusted at all, the current implementation does not yet implement an economic model for determined how much resources should be dedicated to a particular instigator. Instead the first one to reserve an acceptable plan will get the resources.

Once an action plan has been entered, it will hold the resources it needs reserved until 1) the plan is modified or cancelled, 2) the work is completed, or 3) the action is not started in time (its method message arrives late or does not arrive).

When a method message arrives, it contains the action ID ion a SOAP header. It is a simple lookup to find the correct action from the schedule. A continuation is created and associated with the action. Once the trigger driven state machine makes the action runnable (it might be already), the middleware creates a thread to execute the continuation. It creates the thread in a suspended state and copies the stack frame from the continuation and sets the link register to point to an activation completion routine. It then uses the scheduling information in the action to set a time constraint for the new thread and makes it run. This way the pre-existing constraint scheduler (presented in [2]) handles the low-level scheduling work.

Once the method call completes on the activation thread, it returns to the middleware and uses the serializer to send a reply message to the appropriate place. The receiver of the reply message treats it like a service call so the same code is used. The only difference on a client is that a pre-existing continuation with a pre-existing thread is used. The reply message carries the continuation ID in its SOAP header.

After the reply message is sent, any triggers are triggered and resource consumption is determined. An action’s trigger list is walked and the state machine of any named action is advanced and associated continuations executed—triggers are only allowed within a single task within a single node. Meanwhile the low-level scheduler keeps track of resource use and that information is propagated to the sampler module that will, according to the sampling schedule, send the resource and timing information to the instigator planner for quality control and adaptation use.

The network stack uses its own time reservations that are excluded from application use. It is worth noting that there are no starvation gaps between the network stack and the final transmission of the reply message. Deadline driven execution is done at all points and the service thread atomically receives the time constraint from the action plan. The low-level scheduler handles priority inheritance so multiple threads do not starve each other.

3.4. Sampling and Statistics

The instigator receives quality control sampling from the worker nodes. The instigator planner keeps track of all the action plans (tasks) that it has instigated. The sampling messages contain the action ID and the resource and timing information. For each task, the planner maintains a stochastic distribution function. It integrates the new samples into the distribution but uses a negative feedback loop to maintain stability. The samples are maintained as a simple array of a fixed number of samples. A median is calculated and for each sample a standard deviation s calculated. The negative feedback is achieved by inserting a number of old <median, variance> pairs into the array.

The statistical module assumes the distribution is a normal distribution. This simplifying assumption might be incorrect in some applications and more sophisticated match could be used. However, measurements indicate even the simple distribution yields good predictions.

The confidence interval is calculated by integrating the distribution up to the desired application specified probability. Since the distribution is a normal distribution, the integral is independent of the samples and is pre-calculated offline into another array. This precalculated integral is scaled to the observed distribution using the median and deviation.

3.5. Planner

The planner is a middleware component used at the instigator to drive and monitor a particular task. The planner maintains a list of known behavior patterns, which can be extended. The planner also keeps track of all tasks (instances of the pattern) that it has orchestrated. It creates the tasks upon application request by instantiating all the unknowns in a pattern. A discovery process is used to resolve node references into transport addresses. The statistics module is used to cache temporal information. On the very first run the temporal information can be provided by the application or come from a modeling tool. Alternatively the planner will use a guess, an overly large estimate that later is adapted and shrunk to reality. When the initial uncertainty is expected to cause problems, the application service could implement a ‘dry-run’, where all the work is done without actually affecting anything.

The planner periodically scans the task list to make sure that sampling messages have been received correctly—otherwise the application is alerted that the service is misbehaving. The planner also periodically checks whether any tasks’ plans need to be adjusted or reservations renewed.

The planner negotiates with worker nodes to reserve the resources needed by a plan. If any necessary resources are not available, the application is notified so it can take evasive action.

4. Performance

We measured the performance of a demo implementing the scenario in figure 1 on Atmel eb63 boards [12]. They have a 25MHz ARM7 [13] microcontroller. The demo is fully functional and an early version was shown at the Microsoft Faculty Summit in July 2004.

We observe that the entire system can run on a microcontroller with 256KB of ROM and 32KB of RAM. The specific numbers are listed in figure 5. The working of the stochastic planner was estimated through sampling. A simple test method does 20000 multiplications. Starting with no information the planner uses the application guess.

BASE	25356
DRIVERS	13108
NET	77624
XML	8980
SOAP	38336
RTSOAP	14284
APP	43688
CRT	13484
TOTAL	234860

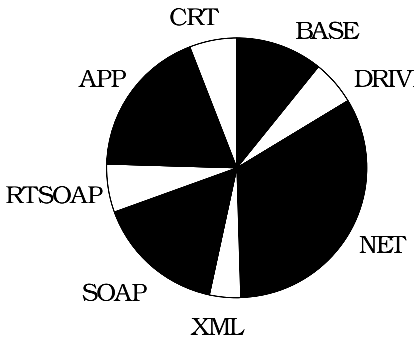


Figure 5, Code size of the components used – in bytes.

Once the planner gets real samples it uses them with smoothing between each step. The calculation times include formatting and sending the reply message. Figure 6 has the numbers. The estimate is produced by the live planner, while the mean and deviation have been calculated offline for reference from the raw measurements.

Step	Estimate 95% conf	Measured mean	Standard deviation	Confidence 95%	99%
1	339	337	1.7%	1.0	1.4
2	341	337	1.6%	1.0	1.4
3	346	337	1.8%	1.0	1.4

Figure 6, Time measurement and prediction of a CPU intensive task – times in milliseconds, 32 samples per iteration on embedded microcontroller board. The confidence number indicates the extra time allocated for jitter. Fixed point integer arithmetic rounds the number up slightly.

Since the low-level scheduler did not produce much jitter, the test was also executed on a PC running WindowsXP with the middleware stack on top. Running without an underlying real-time scheduler introduces more uncertainty but the planner still deals with it correctly and produces a larger confidence allocation to cope with the increased jitter. As the CPU is faster a million multiplications is done each time. From a steady state the number of calculations is dropped to half. Figure 7 shows how the planner adapts to the drop.

5. Related Work

Industrial quality control has been necessary since the beginning of the industrial revolution and the statistical methods are a well-established field of mathematics [7]. This paper extends that tradition into the relatively young area of real-time scheduling.

XML [11] Web services and SOAP [9][10] were originally developed to solve the e-business interoperability problem. Embedded Web Services were presented in [2]. This paper extends them to real-time.

Time-triggered spontaneous messages and conventional service methods are separated into two separate method types in [4]. While this distinction is a step forward in well-defined abstractions, the trigger mechanism and wildcard rendezvous capture both and merge them into a single higher-level abstraction.

Step	Estimate 99% conf	Measured mean	Standard deviation	Confidence 95%	99%
1	126	123	6.4%	1.9	2.5
2	124	120	14%	4.2	5.5
3	69	55	2.1%	2.8	3.7
4	58	55	2.9%	3.9	5.2

Figure 7, Time measurement on PC in milliseconds. After the steady state at step 2, the workload is cut in half and the estimate adapts to the new load.

The planned actions in this paper are an extension of constraint based schedulers [4]. The constraints themselves are an extension of Earliest Deadline First (EDF). CBS allows some prediction on what jobs might be executable since tasks can declare an estimate on how long their work will take. The main limitations in CBS are that tasks can only be declared once the actual work is done and there are no provisions for repetitive work. Instead planned actions can be linked and repeated through triggers and do not have to be ready to execute.

There have been many attempts on dealing with concurrent repetitive tasks. The common type of solution is to reserve "x% of CPU in y second interval". This is fine for simple behavior but cannot effectively deal with 1) mix of repetitive and one-time jobs and 2) cases where the work needs to be done within a constrained sub-interval. The robot scenario in the introduction is one example. For this reason the planned actions do not attempt to over-compress the scheduling information but instead lets the behavior to be described in terms that are as precise as is natural to the application. The scheduler packs the actions in a reasonable manner (given that a perfect packing is NP-complete). Optimizing schedules has been explored in e.g. [6].

6. Conclusion

This paper introduced real-time SOAP and a statistical planned action paradigm as well as several new mechanisms that made the implementation possible and applications easier to write.

The authors conclude based on the implementation that it made sense to extend industrial quality control paradigms to real-time scheduling. Sampling driven statistical scheduling correctly predicts future resource needs in the sample applications. Using XML Web Services (SOAP) in microcontroller systems is viable and makes interoperation issues more manageable.

References

[1] Johannes Helander, Alessandro Forin, "MMLite: A Highly Componentized System Architecture," in *the 8th ACM SIGOPS European Workshop*, September 1998.

[2] Alessandro Forin, Johannes Helander, Paul Pham, Jagadeeswaran Rajendiran, "Component Based Invisible Computing," in *the 3rd IEEE/IEE Real-Time Embedded Systems Workshop*, London, December 2001.

[3] Johannes Helander, Yong Xiong, "Secure Web Services for Low-Cost Devices", in *8th IEEE International Symposium on Object-oriented Real-time distributed Computing* May 18-20, 2005, Seattle, Washington.

[4] K. H. (Kane) Kim, "Basic Programming Structures for Avoiding Priority Inversions", in *Proceedings of ISORC03*, Vienna, Austria 2003

[5] Michael B. Jones, Daniela Roçu, Marcel-Catalin Roçu, "CPU Reservations and Time Constraints: Efficient Predictable Scheduling of Independent Activities", in *Symposium of Operating System Principles,* St-Malo, France, 1997.

[6] K. L. Krause, V. Y. Chen, H.D. Schwetman, "A task-scheduling algorithm for a multiprogramming computer system", in *ACM symposium of Operating system principles,* New York, USA, 1973.

[7] Erwin Kreyzig, "Advanced Engineering Mathematics", 1988, VI edition, pp.1273-1278.

[8] "Web Services Architecture—W3C® Working Draft 8 August 2003," http://www.w3.org/TR/ws-arch/

[9] "Simple Object Access Protocol (SOAP) 1.1—W3C® Note 08 May 2000," http://www.w3.org/TR/SOAP/

[10] "SOAP Version 1.2 Part 1: Messaging Framework—W3C® Recommendation 24 June 2003,"

http://www.w3.org/TR/soap12-part1/

[11] "Extensible Markup Language (XML),"

http://www.w3.org/XML/

[12] "AT91EB63 Evaluation Board User Guide,"

http://www.atmel.com/dyn/resources/prod_documents/DOC1359.PDF

[13] "AT91M63200 Summary, AT91 ARM Thumb MCU,"

http://www.atmel.com/dyn/resources/prod_documents/1028S.PDF

The source code for most of the software described in this paper is available at http://research.microsoft.com/invisible