SLA-Driven Distributed Application Development

Vinod Muthusamy
Department of ECE
Toronto, Canada
vinod@eecg.toronto.edu

Hans-Arno Jacobsen
Departments of ECE and CS
Toronto, Canada
jacobsen@eecg.toronto.edu

ABSTRACT

The management of Service Level Agreements (SLA) in the development of business processes in a Service Oriented Architecture (SOA) often requires much manual and error-prone effort by all parties throughout the lifecycle of the processes. The formal specification of SLAs into development tools can simplify some of this effort. In particular, the runtime provisioning and monitoring of processes can be achieved by an autonomic system that adapts to changing conditions to maintain the SLA's goals. A cost model allows the efficient execution and monitoring of processes, based on a declarative, user-specified optimality function. Experiments demonstrate that the system can indeed adapt to changing workload conditions, saving roughly 70% of the network bandwidth in one particular experiment.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Algorithms, Design

Keywords

Service level agreement, service oriented architecture

1. INTRODUCTION

In a Service-Oriented Architecture (SOA), distributed applications are built by orchestrating reusable services using high-level workflows or business processes. The complexity of developing and maintaining these processes is addressed by SOA development cycles that identify the roles of participants at each stage, and SOA development suites from IBM, Sun or Oracle. However, the development, administration and maintenance of a business process still requires much manual effort that can be automated. In particular, the non-functional goals of a business process, often expressed as Service Level Agreements (SLA), need to be manually considered at each stage of the development process.

This paper presents a vision to achieve end-to-end SLA management by facilitating the various stages of business process development using formally encoded SLAs. We argue for a distributed architecture for the execution of business processes, and develop a model to control the provisioning of business processes in this architecture based on high-level goals that can be specified independently of the processes' implementation details.

The main contributions of this paper are (i) a vision of how formally specified SLAs can simplify the end-to-end SOA development cycle, (ii) the design and development of a cost model and distributed architecture to execute business processes, and to dynamically provision resources based on high-level goals, and (iii) an evaluation of the distributed execution engine.

MW4SOC '08, December 1, 2008, Leuven, Belgium

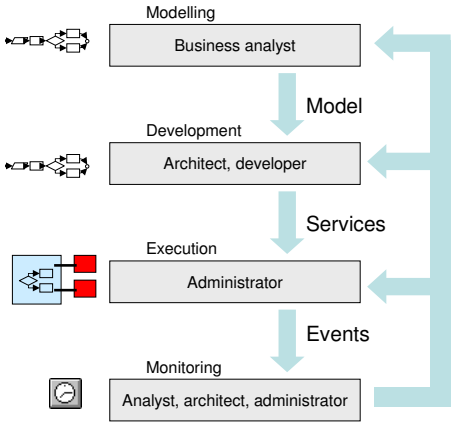


Figure 1: SOA Development Cycle

2. SERVICE LEVEL AGREEMENTS

This section outlines a typical SOA development cycle, and follows with a vision of the benefits to this process of managing SLAs within the development tools.

2.1 SOA Development Cycle

The SOA development cycle as illustrated in Figure 1 consists of the modelling, development, execution, and monitoring stages. Each stage is concerned with a different level of abstraction and is performed by the indicated roles, each of whom have varying expertise and concerns.

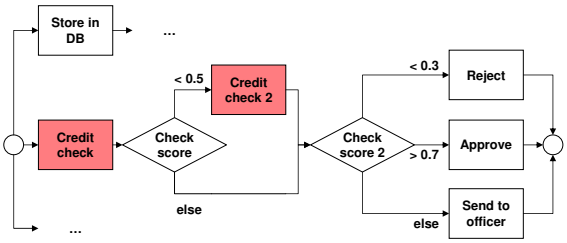


Figure 2: Loan Application Process

Consider the simple business process fragment in Figure 2 in which a loan application is processed. This process first calls an external credit check service to determine the applicant's credit rating, with a second more refined credit check service used in the case of a low rating. This rating is used to determine if the application should be approved, denied, or processed further by a loan application officer.

In the modelling stage of the development cycle, the business analyst would define the above process, abstracting from technology, infrastructure, and implementation details. The result of the modelling stage is an abstract model of the process represented in BPEL or a proprietary process language. This representation is imported into the development stage, where architects and developers break the model into development artifacts such as services along with their interfaces, and implement the required business logic. The result of this stage is a set of deployable components. These services are deployed in a runtime environment that is managed by an administrator responsible for ensuring physical resource provisioning sufficient for the goals of the deployed services and processes. Often it is desirable to monitor the execution of the processes by tracking metrics on the state of the executing system. The metrics gathered can be aggregated and presented to the stakeholders in the preceding development stages. For example, the business analyst may be interested in high level metrics such as the number of times the second credit check is required. On the other hand, the system architect may be interested in lower level metrics such as the processing delays of the individual credit checking services, while the administrator would be concerned with system performance bottlenecks such as network congestion or processor utilization.

In the modelling stage, high level, declarative goals are specified by the analyst, such as the throughput requirements or the cost constraints on the process. These goals are formalized into Service Level Agreements that specify a contract between the service provider and consumer. SLAs can be represented at different levels of abstractions, and may simply be a document at the modelling stage. These SLA documents are passed down the chain of the development process, with each stakeholder being responsible for ensuring conformance to the SLAs. The architects and developers interpret and ensure that the services developed conform to the SLAs. During execution, SLA conformance is often achieved by over-provisioning resources and manually tuning the system. Finally, monitoring subsystems are instantiated to verify that the SLA goals are met, and that violations are reported to the appropriate parties. These violations are manually addressed by changes to the process, redevelopment of services, or provisioning of resources.

2.2 Integration of SLAs

If SLAs are integrated into the tools and translated into execution and monitoring models, violations can be tracked more quickly and resources can be provisioned on demand in response to or in anticipation of violations. For example, SLAs at the modelling stage can be mapped to lower level requirements on the services developed and resources provisioned, and translated to metrics that need to be monitored to observe SLAs violations. Furthermore, adaptations on the process itself or its resource provisioning can be performed automatically at runtime to maintain the SLA goals.

Several runtime adaptations can be performed to maintain SLAs. *Dynamic service selection* can occur whereby the most appropriate services are chosen from a catalog of available services. For example, a fast or cheap credit check service can be used based on the SLA requirements. As well, *monitoring* can be optimized by only observing those metrics that are relevant to the SLA. As we outline in Section 3.3, the *distributed execution* of business processes becomes feasible by automatically assigning portions of a process to strategic locations in the system. Furthermore, dynamic *resource allocation* can take place to ensure the process has sufficient resources (CPU, bandwidth, etc.) to maintain the SLA despite changes in the process load. Finally, the Enterprise Service Bus (ESB)that underlies the SOA can be reconfigured to satisfy the SLA. It is important to emphasize that the encoding of the SLAs into the SOA tools in a machine understandable format makes it possible for these runtime adaptations to take place dynamically and without human intervention.

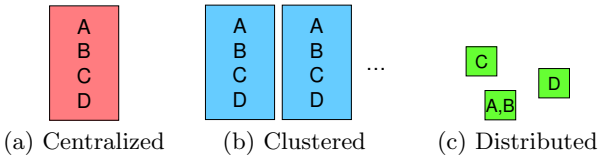
Incorporating SLAs into the SOA development tools *simplifies* the specification of SLAs since the analyst can declaratively specify business process goals without detailed knowledge of the underlying technologies. Additional *flexibility* is achieved by allowing the developers and administrators to make design decisions without having to be as concerned about SLA violations since the tools can perform some of these tasks. As well, the analyst can change the SLAs and be confident that these revisions will be propagated and enforced throughout the development stages.

3. SYSTEM ARCHITECTURE

Business process execution engines are typically centralized systems in which one node executes and manages all instances of one or more business processes. To address scalability, the centralized engine can be replicated and the process instances balanced among the replicas. In this work, we take a fundamentally different architectural approach whereby even individual process instances are executed in a distributed manner. The benefits of this architecture include scalability, in-network processing, and fine-grained use of IT resources. We further describe and compare the various business process execution architectures below.

3.1 Execution Architectures

Centralized: The simplest business process engine consists of a single execution engine as shown in Figure 3(a). This centralized engine is responsible for executing and managing all concurrent instances of the processes deployed on it. The advantage of such an architecture is its simplicity in terms of deployment and management. However, as the resources in such an architecture are fixed, the system may not scale with the complexity of processes and the number of



(a) Centralized (b) Clustered (c) Distributed

Figure 3: Process execution engine architectures

process instances. Also, as the single execution engine represents a single point of failure and may not be appropriate for the execution of mission critical processes. Furthermore, inter-organization business processes may have no obvious choice for a central coordinator.

Clustered: To address scalability and fault-tolerant, a cluster of execution engines can be deployed. In this architecture, illustrated in Figure 3(b), each engine in the cluster is essentially a replica of the others, and can execute a complete business process. A call to a business process P is first sent to a load balancing component (not shown in the figure), which forwards the call to one of the engines E in the cluster, based on some criteria such as ensuring a balance of load across the cluster. At this point, engine E creates an instance of process P and is responsible for executing the instance until completion. Some systems support the ability to add and remove engines to the cluster as the load varies. A clustered execution architecture can be scalable and does not suffer from a single point of failure. However, process *instances* are still executed in a centralized manner, and control and data is still concentrated in the cluster. Consider the case of a data-intensive process such as a scientific workflow that transfers and operates on large volumes of data. In a clustered architecture, the data needs to be transfered to the cluster before it can be operated on by the process. In a more flexible deployment it would be possible to move the portions of the process that operate on the data closer to the data source thereby reducing the time and network costs incurred in having to transfer the data.

Distributed: This paper opts for an execution engine in which processes themselves can be distributed. As shown in Figure 3(c), a process is first decomposed into tasks, which are then assigned to various execution engines in the system. In a BPEL process, the tasks can be the individual BPEL activities. To emphasize the fact that these execution engines can be light-weight as they only execute fine-grained tasks, as opposed to complete processes, we refer to the entity that execute tasks as an *agent*. A key benefit of such an architecture is the ability to deploy *portions* or processes close to the data they operate on, thereby minimizing bandwidth and latency costs of a process. For example, for data intensive business processes (e.g., rendering farms, large simulations etc.) it would be possible to deploy only those portions of the process that require access to large data sets close to their respective data sources. Different parts of the process that operate on different data sets can be independently deployed near their respective data sources. This is not possible in a clustered architecture since the entire process instance must be executed by a single engine.

The benefits of the agent-based execution engine architecture are only achieved if the agents are deployed in a strategic manner. This can be a labour intensive procedure that requires knowledge of the system resources, and process characteristics. It may even be a futile exercise if either

Component	Notation
Distribution cost	C_{dist} (distribution overhead)
Message rate	C_{d1}
Message size	C_{d2}
Message latency	C_{d3}
Engine cost	C_{eng} (execution overhead)
Load	C_{e1}
Resources	C_{e2}
Task complexity	C_{e3}
Service cost	C_{serv} (service overhead)
Service latency	C_{s1}
Service execution	C_{s2}
Marshalling	C_{s3}

Table 1: Cost model components

Criteria	Cost function mapping
3s response time	$C_{d1} + C_{d3} + C_{e3} + C_{serv} < 3$
Optimize bandwidth	$min(C_{d2})$

Table 2: Examples of Cost Functions

of these variables changes frequently. It is desirable for the system itself to determine an optimal placement of agents. To achieve this, we develop a cost model below to model the cost of a particular placement of agents. This model is used to compare different placement possibilities.

3.2 Cost Model

The cost model consists of various factors that can influence the agent placement decisions. Some cost factors are shown in Table 1 grouped into *cost components*.

The first component is the *distribution cost* which represents the overhead of distributing a process into small, fine-grained agents. This overhead can be expressed in terms of the bandwidth or latency of the inter-agent communication depending on the desired goal. .

Another important cost component captures the resource usage of an agent on the engine it is executing on. Factors here include the number of concurrent instances an agent is executing, the resource utilization (in terms of processor or memory) of an agent, and the complexity of the task the agent is executing.

The third cost component in Table 1 is the *service cost* which represents the cost of calling external services. This includes the time to call the service (which is a function of the network conditions between the agent and service), and the execution time of the service (which depends on the service provider used to execute the desired service).

A *cost function* based on the various cost components allows flexibility in specifying different goals easily. The cost function specify that an arbitrary weighting of the various cost components either meet a *threshold* or should be *minimized*. In the former case, the process is adapted only when the threshold is violated, while in the latter, process adaptation occurs whenever a more optimal placement is found. For example, Table 2 shows cost functions that ensure that the response time of a process is within three seconds, and that minimize the network overhead of a process.

3.3 Distributed Execution

As discussed earlier, we take a distributed approach to the execution of business processes, whereby individual tasks in a process are executed by autonomous agents which collab-

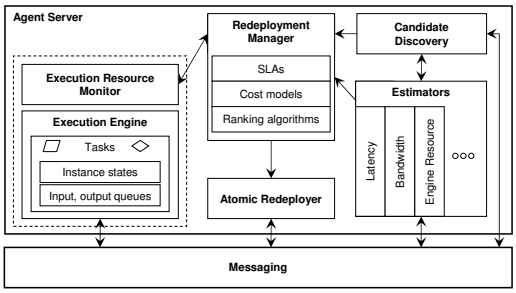


Figure 4: Distributed Execution Engine

orate to execute the original process. The agents execute within a distributed execution engine whose architecture is shown in Figure 4.

In the spirit of the distributed nature of the system, each agent is autonomous in deciding which engine it should execute on and when it should move itself to another engine. These decisions are based on the cost function associated with the process, which is known to each agent in the process. Notably, there is no centralized component that is used to gather statistics, or to make agent placement decisions.

The distributed execution engine shown in Figure 4 consists of a core Execution Engine that provides support services for agents to collaborate among one another to execute a particular business process. A Candidate Discovery component is used to find other execution engines in the system. In the current implementation, the discovery component returns all immediate neighbours of an engine and uses a random walk to find a random set of distant engines. The discovered candidates are periodically probed by the Estimator components to gather various statistics. The Redeployment Manager computes the cost function for each agent executing in the engine, and determines if a more optimal placement of the agent is available among the known candidate engines. Finally, agents that are to be moved are redeployed using the Atomic Redeployer component which is responsible for ensuring that the movement of the agent does not affect the execution of the process. Briefly, the redeployer pauses the triggering of new instances of the agent, transfers the agent state to the new engine, rebinds the agent to its successors and predecessors in the process, and resumes the execution of the agent.

3.3.1 Redeployment Manager

The Redeployment Manager maintains for each agent A_i the agent server is currently hosting, the cost function $F(A_i)$ associated with the agent, a running average of the cost $C(A_i, S_j)$ imposed by the agent were it hosted by agent server S_j, and the agent servers where agent A_i's predecessors and successors are hosted. For convenience, the cost of deploying A_i at the current server is denoted as $C(A_i)$. The cost $C(A_i)$ has two different interpretations depending on the type of cost function. For threshold functions, $C(A_i)$ is the *accumulated* cost by all agents from the beginning of the process to the current agent, whereas for minimum or maximum cost functions, $C(A_i)$ is the *local* cost of the agent. An example should make the reason for the difference clear. Consider a cost function to minimize the message latency of a process: $min(C_{d3})$. In this case, the local cost of an agent is the latency of communicating with its predecessors and successors. To minimize the overall latency, each agent should attempt to minimize its local latency cost. On the other hand, for a cost function that requires the message latency to stay below a threshold, such as $C_{d3} < 10$, it is necessary to keep track of how much each agent contributes to the overall latency of the process. The local latency cost of each agent must be accumulated as the process flow executes. To achieve this, for agents associated with threshold cost functions, messages between agents are annotated with the accumulated cost.

The running average of the cost $C(A_i, S_j)$ of an agent is computed and maintained by the Redeployment Manager based on information from various Estimators or the Execution Resource Monitor. The Redeployment Manager recomputes the costs $C(A_i, S_j)$ when one of two conditions occurs. When the agent A_i executes (including when it sends and receives messages with its successors or predecessors), the cost is updated for the current server, i.e., $C(A_i, S_{current})$. Likewise, when an estimator updates a metric that is included in the cost function associated with A_i, the cost of hosting the agent at the candidate server whose metric was just updated is recomputed. To facilitate the latter case, each estimator given a list of agents which are relevant to the metric the estimator is computing. Therefore, the estimator will only notify the Redeployment Manager when necessary.

An update of the cost $C(A_i, S_j)$ may reveal a better placement for agent A_i. Every update to the cost of an agent initiates a call to the $CheckDeployment(A_i)$ function to find a more optimal deployment. The algorithm differs based on the cost function associated with agent A_i.

If the cost function requires the cost to be minimized, then the algorithm finds the server $S_{min} \in S$ such that $C(A_i, S_i)$ is minimized across all $S_i \in S$ where S is the set of known candidate agent servers. The agent A_i is then moved to agent server S_{min}. To avoid frequent redeployment, an agent is redeployed only if the improvement in the cost exceeds a given threshold $T_{benefit}$ and if the agent has not been redeployed for some time duration $T_{duration}$. The values $T_{benefit}$ and $T_{duration}$ have default system wide values that may be overridden by specific ones for each cost function.

If the cost function associated with agent A_i is a threshold function, a check is made to see if the accumulated cost $C(A_i)$ exceeds the threshold. If the cost is still within the threshold, nothing further is done. Otherwise, the $CheckDeployment()$ function finds the agent server S_{min} that results in $\min_{S_i \in S} C(A_i, S_i)$, and redeploys agent A_i to agent server S_{min}. Now it may be that $C(A_i, S_{min})$ still exceeds the required cost function threshold, in which case a message is sent to the predecessor servers of agent A_i to force them to redeploy. Notice that these predecessors would not have normally chosen to redeploy because their accumulated cost is still within the threshold. This "push back" by agents to force a redeployment of their predecessors will occur repeatedly as long as the optimal placement of the agent is not sufficient to satisfy the cost function threshold.

3.3.2 Estimators

Estimators compute various metrics necessary to rank possible placements of agents and determine the optimal placement. To avoid unnecessary estimations, the Redeployment Manager enables estimators only if there is a locally hosted agent whose cost function depends on the metric computed by the estimator. For example, if there is an agent A_i whose

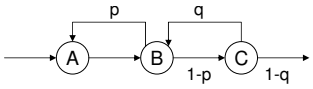


Figure 5: Business Process with Loops

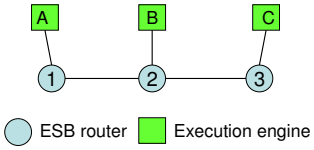
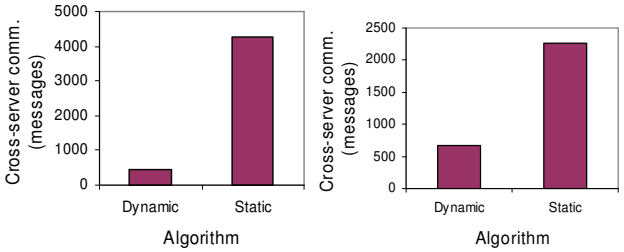


Figure 6: Experimental Topology



(a) Scenario 1 (b) Scenario 2

Figure 7: Message Cost for Process in Figure 5

cost function is to minimize message latency, then the Latency Estimator would be enabled but not the bandwidth estimator. Each estimator is also provided with additional information necessary to perform the estimations. For example, the latency estimator is given a set of the predecessor and successor servers S_N associated with relevant agents A_i. The redeployment manager then estimates the latency between nodes in S_N with the nodes S_C discovered by the Candidate Discovery component. The Service Latency Estimator, on the other hand, is provided a list of services invoked by relevant agents and the estimator computes the time to invoke the services from each node in S_C.

3.3.3 *Atomic Redeployer*

The movement of an agent A_i from agent server S_1 to agent server S_2 as determined by the Redeployment Manager is carried out by the Atomic Redeployer. The challenge is to move an agent without disrupting the process and to ensure that failures during movement do not leave the system in an inconsistent state.

The movement is modelled as a transaction consisting of a $move(A_i, S_1, S_2)$ operation. If the transaction aborts—perhaps because S_2 refuses to accept agent A_i—the agent must remain at server S_1. Otherwise the transaction commits, and the agent must be instantiated at S_2 and deallocated from S_1. In either case, the predecessors and successors of agent A_i should be unaware of the movement, no messages must be lost, and each message should be delivered to the agent instance at S_1 or at S_2 but not both. These requirements and others have been formalized in detail, and the algorithms to achieve atomic movement satisfying these requirements have been developed, their correctness proven, and their performance quantified [5].

4. EVALUATION

In this experiment, we use the business process shown in Figure 5 deployed to a set of execution engines in Figure 6. The process is designed to model a business process with time varying branch probabilities. Also, while the process only consists of three tasks, the looping constructs in the process results in many tasks being executed in the course of a process instance. The agents associated with the tasks in the process are initially deployed as shown in Figure 6.

In this process, the branch probabilities of tasks B and C are varied and the reactions of the redeployment algorithms are observed. In the experiment, the SLO associated with the process seeks to minimize the bandwidth used by the process (see Table 2). Consequently, the metric we observe is the number of messages sent between the execution engines. Note that messages between agents deployed on the same engine are not counted in this measurement.

The metric to minimize bandwidth results in all agents being deployed on the same execution engine regardless of the workload. To make the experiment more interesting, we fix the agents associated with tasks A and C to their initially deployed engines in Figure 6, and allow agent B to move freely to any engine. This represents the case where certain tasks must be executed on engines owned by a particular department for administrative or security reasons.

Figure 7(a) shows the results of an experiment in which the branch probabilities of the process are fixed at $p = 0.9$ and $q = 0.1$ and 100 instances of the process in Figure 5 are invoked. Note that this workload is biased toward the loop involving tasks A and B. We observe that the system reconfigures itself from the initial deployment in Figure 6 to one where agent B runs on the same execution engine as agent A. In Figure 7(a), this results in the dynamic algorithm having about 10% of the message cost of the static algorithm in which reconfiguration is disabled and the agents remain in the initial deployment in Figure 6. The point here is that it is not necessary to manually deploy agents in a strategic manner — a possibly complex task — but to allow the system to configure itself.

In another experiment, the branch probabilities are varied, starting out with $p = 0.9$ and $q = 0.1$ for the first half of the experiment, and changing to $p = 0.1$ and $q = 0.9$ for the second half. Therefore, the workload initially results in a lot of communication between tasks A and B, and then becomes biased towards tasks B and C. This time, the deployment starts with agent B at the same engine as agent A, which is the optimal placement for the initial branch probabilities in this experiment. Again, we observe that the dynamic algorithms keeps agent B in the initial optimal engine, and only when the branch probabilities change, the system moves agent B first to the middle execution engine and then to the one with agent C. The results in Figure 7(b) confirm that the dynamic reconfiguration algorithms adapt to the changing workload to achieve the desired goal with less than 30% of the message cost of the static case. A notable point about this experiment is that because the conditions of the system change over time, there is no optimal static deployment. Dynamic reconfiguration is required to achieve the best results.

5. RELATED WORK

Distributed workflow processing has addressed scalability, fault resilience, and enterprise-wide workflow manage-

ment [4, 11, 9]. Heinis *et al.* [4] develop a self-optimizing distributed workflow engine. However it does not support flexible optimization criteria as in this paper, and takes a centralized approach to monitoring and reconfiguring the workflow deployment. This differs from the design in this paper which is fully distributed, but is less likely to find globally optimal solutions. A behaviour preserving transformation of a workflow into an equivalent partitioned one is described in [9] and realized in the MENTOR system [11]. This is complementary to our work since we operate with the original business process model without analysis.

Distributed stream processing engines [6, 1, 2, 10] install a set of *operators* in the network to process streams of data and execute SQL-like queries over the streams. However, systems such as Borealis [1] use a proprietary query language and do not support loops in the query network, which makes it unsuitable for business processes. None of these approaches consider the planning or scheduling of resources based on SLAs determining higher-level business goals. In IFLOW [6] nodes are organized in a cluster hierarchy, with nodes higher in the hierarchy assigned more responsibility. This differs from our completely distributed architecture. As well, IFLOW associates utility with links rather than the nodes, which makes it difficult to express requirements such as minimizing sever load. These engines also do not support service selection by, for example, requiring a task to select a service with minimum latency.

While stream processing engines may bear some architectural resemblance to a set of agents executing a business process, there are issues related to business process execution that are not easily handled by stream processing engines. First, the stream processing work above is based on proprietary languages, not an industry standard such as BPEL. More significantly, a business process is conceptually not simply a data stream. There are notions of process instances and the accompanying state and isolation semantics that are not required in streams.

In addition to the semantic differences between processes and streams, process distribution in our approach differs from the above work by exploiting an underlying messaging substrate. Like [6], our agents are decoupled by communicating using content-based names instead of network identifiers. In addition, we plan to utilize the advanced features of our messaging substrate, developed in earlier research, to offload some of the agent processing to the network [7]. This simplifies the agents, and allows the network to optimize this processing logic.

This paper builds on our prior work on efficient messaging systems [3, 7]. We also leverage our research on the NINOS system [8], which develops a distributed BPEL execution engine, and provides a platform for task decoupling, dynamic reconfiguration, system monitoring, and run-time control.

6. CONCLUSIONS

The development of business processes in Service Oriented Architectures involves several stages in which different actors concerned with different aspects of the process contribute to the realization of the final process. Each actor must be aware of and fulfill any goals associated with the process. In this paper we present a vision where these goals are formally represented in an SLA specification in the development tools, and used to simplify each stage of the development cycle from the modelling of the process, through the development, to the deployment, execution and runtime monitoring of the process.

A distributed architecture is proposed for the execution of business processes in which light-weight agents collaborate to execute a larger process. This architecture affords scalability by allowing more fine-grained resource allocation, and the ability to strategically move computation close to the data it operates on. To simplify the management of this process execution architecture, a cost model is developed to allow goals, such as minimizing bandwidth resources and process response times, to be independently specified on the executing process, and algorithms are devised to redeploy the executing process to satisfy the specified goals.

Evaluations support the ability of the system to repeatedly adapt to changing runtime conditions to achieve a declaratively specified goal. In one workload, the system was able to save about 70% of the bandwidth by adapting a process compared to an initially optimal, but static deployment.

We are evaluating the system further with more realistic and complex processes, workloads and SLAs. We also plan to compare the trade-offs between the non-optimal but distributed reconfiguration algorithms presented with centralized one with global knowledge of the system configuration.

Acknowlegements: The authors would like to thank IBM CAS Toronto for their support towards this research.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[3] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of ACM SIGMOD*, 2001.

[4] T. Heinis, C. Pautasso, and G. Alonso. Design and evaluation of an autonomic workflow engine. In *ICAC*, 2005.

[5] S. Hu, V. Muthusamy, G. Li, and H.-A. Jacobsen. Transactional mobility in distributed content-based publish/subscribe systems. Technical report, University of Toronto, Apr. 2008.

[6] V. Kumar, Z. Cai, et al. Implementing diverse messaging models with self-managing properties using IFLOW. In *ICAC*, 2006.

[7] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, 2005.

[8] G. Li, V. Muthusamy, and H.-A. Jacobsen. Ninos: A distributed service oriented architecture for business process execution. Technical report, University of Toronto, Nov. 2007.

[9] P. Muth, D. Wodtke, J. Weisenfels, A. K. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *JII*, 1998.

[10] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[11] D. Wodtke, J. Weisenfels, et al. The Mentor project: Steps toward enterprise-wide workflow management. In *ICDE*, 1996.