XCHANGE: High Performance Data Morphing in Distributed Applications

Jay Lofstead and Karsten Schwan

College of Computing, Georgia Institute of Technology

{lofstead, schwan}@cc.gatech.edu

Abstract

In both high performance and enterprise applications, it is common for components to generate, exchange, process, and store or display large volumes of data. A key problem for such exchanges is the mismatches of data being generated with the data required by communicating software components. Such mismatches arise from natural differences in the data representations used by different components and for acquisition or display. They are due to the need to customize or personalize data for certain devices or end users. This paper describes automated methods and associated generative tools for correcting such mismatches using overlay networks to connect data producers with consumers. These data morphing methods automatically generate data transformation codes from declarative specifications 'just in time' and 'in network', i.e., when and where needed. Runtime code generation takes into account the current nature of the data being generated, the current needs of data sinks, and the current resources available in the overlay connecting sources to sinks. In addition, it can consider the shared requirements of multiple consumers to reduce redundant data transmissions and transformations. Dynamic, 'in network' data morphing is realized with the XCHANGE toolset, which can generate transformation codes both from high-level declarative specifications like MathML and with methods that automatically correct for type mismatches. XCHANGE is integrated with ECho, the high performance event-based middleware and it is applied to both high performance and enterprise applications. Results show that for remote visualization of the data created by a high performance simulation, dynamic data morphing can better match the large data volumes being exchanged to available network resources. For an enterprise application in the healthcare domain, dynamic data morphing demonstrates the importance of generating code to reduce server loads.

Keywords: Data Transformation, Dynamic Data Morphing, Dynamic Code Generation, Overlay Networks, XML, Declarative Specification, Distributed and Enterprise Applications, High Performance Data Exchange.

Technical Areas: Internet Computing and Applications, Autonomic Computing, Data Management

1. Introduction

A large class of service-oriented applications uses distributed components to capture, process, and display data in real-time across heterogeneous underlying execution platforms [47, 48, 49, 50]. A specific example is a scientific collaboration in which data is streamed from a running simulation to multiple end users. Here, large data volumes are accompanied by substantial processing demands when different end users require individualized views of simulation outputs. Another example is in the healthcare domain, where a hospital must share patient data across many distributed subsystems [35]. Here, scalability concerns limit the overheads acceptable for the data processing and conversions needed to match data representations to the diverse needs of individual subsystems.

This paper addresses a general issue for the data-driven, service-oriented, distributed applications described above: how to efficiently deal with dynamic mismatches in the data formats output by one application component (i.e., data source) vs. the inputs required by another (i.e., data sink). Our XCHANGE approach to dealing with such mismatches performs runtime, 'in network' *data morphing*, that is, it dynamically constructs and automatically populates with data transformation codes overlay networks connecting sources to sinks. Runtime code generation and deployment are based on meta-information that includes (1) type information based on which data mismatches between outputs and inputs are diagnosed and corrected via automatically generated code and (2) transformations explicitly described by user-specified, meta-level instructions from which efficient binary codes are generated at runtime, whenever or wherever needed. Dynamic compilation and code deployment also capitalize on the data streaming nature of our target applications by eliminating repetitive data transformation actions and sharing selected transformation results across multiple data sinks.

XCHANGE addresses data mismatches that range from row- vs. column-major data layouts and their associated impacts on the performance of parallel applications [21] to changes in time or length scales for data emitted vs. consumed by multi-scale modeling applications [26] to resolution changes or per-client requirements for data output by some simulation model vs. what is displayed by an end user device (e.g., see [1] or [27]). It can also be used to deal with mismatches in data acquisition, as with satellite data repeatedly processed until ingested by some simulation model or sensor data processed, filtered, and transformed prior to being used for some command and control function [28]. Finally, XCHANGE has been shown useful for applications where privacy or proprietary concerns require data to be filtered or transformed prior to being released to another software component [29, 60].

A specific set of data mismatches studied in this paper occurs in the SmartPointer [1] framework for real-time scientific collaboration, which can create per-user and per-device custom views of scientific data. For the molecular dynamics (MD) simulation data interactively viewed with SmartPointer, some users may be interested in viewing copper atoms and their positions, for instance, whereas others are interested in the bond forces occurring in the simulated material. Using the traditional client-server approach to distributed computing, a straightforward approach to viewing such data is to send all of it to each sink and then have each sink perform the processing necessary to implement its specific data view. For the large data volumes emitted by modern MD applications, this will generate additional and needless data transport and processing actions. The alternate approach of moving data conversion operations to the source has been shown useful [45], but it is limited by the implied strain on source-side host CPU resources [30, 46] and/or the degrees of data expansion experienced at the source. XCHANGE affords us the flexibility to combine the generation of data transformations with their 'in network' deployment. That is, appropriate code generation makes viable the deployments best suited for current resource conditions: into sinks, into the overlay, or into sources. As a result, by combining code generation with deployment and using real-time monitoring, data morphing is customized to overlay characteristics like the processing resources on certain overlay nodes and the network resources between these nodes. The outcome is improved scalability and/or performance [13], which XCHANGE evaluates with application-specific metrics. One metric used in this paper creates data conversions and maps them to overlay nodes to reduce total network usage [31, 32]. Others might minimize resource consumption [33] or some global metric like end-to-end delay [13].

The XCHANGE toolset and implementation provide the following new functionality: (1) high level descriptions of data transformations processed by a transformation descriptor parser, (2) an optimizer that carries out cross-request optimizations on the code being generated, (3) a code distributor for mapping the optimized graph of transformations onto available overlay nodes, and (4) a transformation deployment daemon -- controller -- able to dynamically place transformation codes into the overlay. XCHANGE can be associated with any overlay-based messaging or data distribution middleware. The middleware used in our experiments is EVPath, a subset of the ECho high performance publish/subscribe infrastructure [6]. EVPath provides the overlay abstractions and control interfaces required by toolsets like XCHANGE offering control methods for overlay management to dynamically create or delete new nodes, to install new data transformers or filters, to deploy code that changes how messages are routed across the overlay, and to acquire monitoring information about the underlying distributed execution platform.

Furthermore, targeting the high performance domain, the current implementation of XCHANGE with EVPath uses the Portable Binary I/O format (PBIO) [2] to describe the structure of the binary data being moved, somewhat like the 'trivial' binary encodings of XML in [52] describe the structure of XML records in binary forms. Addressing the high performance domain, however, EVPath also uses PBIO binary data encoding for its actual data exchanges combining or replacing such exchanges with XML schema-based data descriptions [10] (via the XMIT [8] library) only when or if needed by participating components.

Jointly, XCHANGE and ECho offer functionality much beyond that of network-level multicast implementations like MBone [5] and they have been applied to a wide range of underlying execution platforms and network topologies. In these contexts, the advantages of using XCHANGE for implementing distributed data transformations are multi-fold:

- *Efficient data morphing through runtime code generation*. By explicitly describing data transformations, code implementing them can be generated automatically and dynamically in the forms needed for the currently 'best' deployment to distributed overlay nodes. When bandwidth is limited, for example, generation and deployment can be optimized to minimize unnecessary data transmission. Furthermore, runtime code generation makes it feasible to optimize data transformations (1) across entire affected overlay paths and (2) across all involved data sinks and sources. Sample optimization techniques used in this paper include the recognition and elimination of replicated data transformations and the use of operation dependencies to appropriately distribute operations across different overlay nodes. The outcome of such analyses is an optimized scheme for dynamically evolving data from a source to each of its sinks.
- *Dynamic, incremental deployment*. XCHANGE implements efficient techniques for incorporating new requests into existing overlays with small impact on the currently running application. Such dynamic deployment also affords us the opportunity to consider reconfiguring an existing overlay network based on changing conditions.
- *High performance.* By directly operating on binary data, XCHANGE-generated codes are applicable to both traditional high performance applications moving large data volumes and to applications that require low overheads for high request volumes. That is, XCHANGE does not require application data to be in an XML format. Rather, it uses XML-based data format descriptions to generate efficient binary codes that operate

on the binary data being described. The motivation is to avoid (1) the overheads caused by runtime XML parsing and (2) the data expansions experienced when using XML data encodings [52].

An interesting attribute of XCHANGE not highlighted here is that its code generation and deployment schemes are replaceable since they are based on explicit meta-information maintained by the middleware about the overlay to which code is being deployed.

This paper demonstrates the utility of XCHANGE for dynamic data morphing with a distributed scientific data visualization, which consists of a single server sending display data to multiple sinks each requiring different data transformations. A deployment scheme optimized to reduce network usage causes a 64% reduction in network usage vs. a sink-hosted transformation model and a 14% reduction over a source-hosted transformation model. A second use of XCHANGE for an application in the healthcare domain focuses on maintaining high end-to-end performance while also offloading the transformation CPU workload from endpoints. One scenario uses a representative 'results' message from a laboratory system to a transactional system where the two end points use different data organizations and have different scaling requirements. By transforming message formats in an intermediate node rather than at the endpoints, tasks can be offloaded from resource-poor or highly loaded end points with an up to 17% improvement in end-to-end performance.

The remainder of this paper is organized as follows. Related work is discussed in Section 2 followed by an overview of the XCHANGE architecture and implementation in Section 3. Section 4 discusses and evaluates the XCHANGE implementation. Experimental evaluations appear in Section 5. Conclusions and future work appear in Section 6 followed by references in Section 7.

2. Related Work

A large set of research and systems uses SQL-like constructs to transform and combine data with an overlay network linking sources and sinks, as in publish/subscribe infrastructures [6, 36, 51], in systems that aim to provide rich data access models like DataCutter [12] and MOCHA [16, 18], STREAM [19], Aurora [20], In-Transit [13], and in formulations of continual query systems [55, 56]. Since data manipulations are expressed as SQL-style operations, the semantic information present in query graphs can be used to automate tasks like dynamic query routing [36], query graph deployment [13], overlay network construction and repository consistency [53], and similar topics. Most such work is complementary to XCHANGE, which attempts to combine the generation of data manipulation codes with the mapping process thereby offering an additional degree of freedom in how data

transformations may be carried out. We share with such systems the assumption that the data being manipulated is represented in some well-structured form rather than as unstructured XML. Borrowing concepts from such research, it would be interesting to make our code generation and therefore our data distribution consistency-sensitive [53, 58, 59] or perhaps even drive it with explicit measures of data similarity [57]. Scalable methods for overlay deployment and management like those described in [13] are not relevant to this paper's experimental results. Here we simply use these methods to experiment with alternative realizations of data manipulations and their mappings to different overlay nodes.

DataCutter [12, 14] delivers client-specific data visualizations to multiple end points, but there are some key differences with our work. First, DataCutter uses C++ code for custom filter functions thereby imposing programming tasks on end users. Second, for automatically generated filters, DataCutter uses an SQL-like 'flat' data model and approach for data selection whereas we use comparatively richer descriptions for both filter and transformation operations. The manipulation codes generated from these descriptions operate on data in the forms with which end users are already familiar. In addition, code generation can optimize similar requests across multiple clients by combining common portions of transformations from multiple sinks into a single operation and/or by splitting transformations for distribution to the most appropriate locations on the paths from sources to sinks.

Many tools use XML syntax to describe querying of data sources including XWrap [54] and YATL [3]. For instance, YATL, built as an extension to the ML language, provides a declarative language for transforming a XML document into a new one based on the query issued. In contrast, we manipulate data in its native binary form using XML at the meta-level to specify the data types and transformations being applied to binary data. Furthermore, our approach permits use of a variety of specification languages including those built on existing standards like MathML [11] and XML Schema [10]. We also do not limit the source formats of data, which may be XML or any other understandable format such as binary data created on the fly.

Our work can also be compared to the programming methods offered by Java-based infrastructures and runtime deployment functionality like Enterprise Java Beans (EJBs) [34]. Of particular note is that EJBs and many similar technologies require a fixed type interface dictating the order and decomposition of necessary transformation operations at design time and the deployment descriptors and interfaces are codified separately from the actual transformations. While our work lacks the richness of EJB implementations, for the applications we consider, that penalty is offset by the fact that the actual decomposition and ordering of transformations can be completely

rearranged, whenever necessary, to account for changes in resource availabilities or new client requests, without any coding changes.

Finally, this work and our previous research with ECho [6] share the ability to dynamically install application-specific stream filters and transformers with systems like Infopipes [17].

3. XCHANGE Architecture and Implementation

3.1 Overview

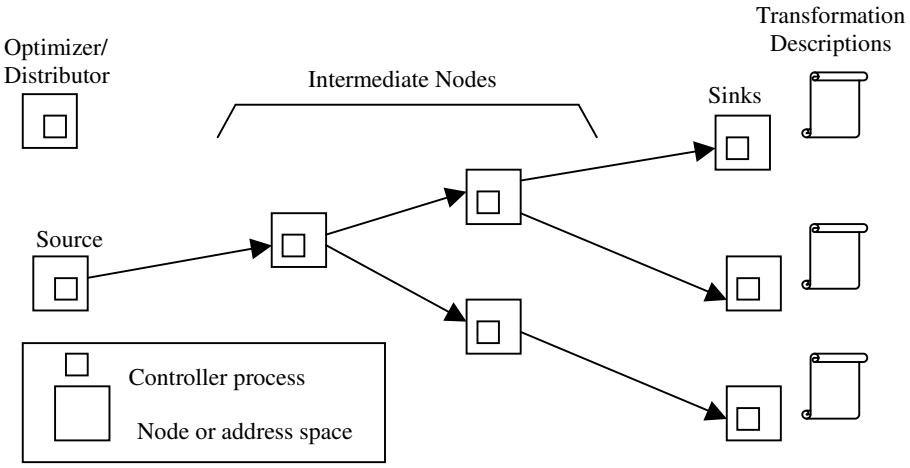

Figure 1: Example Overlay Network

Figure 1 depicts a simple overlay network connecting multiple sources to sinks, where the main component of interest is the 'controller', which interacts with overlay nodes to install new transformations or change existing ones. Data transformations in XCHANGE involve the following components:

- *Source*. A source generates a single copy of a data record for downstream consumption with some well-defined source format.
- *Sink*. Consumes the data in some desired target format.
- *Transformation Description*. Describes the source and sink data formats and the transformation operations necessary to perform the transformation. One of these is required for each sink.
- *Optimizer/Distributor*. A process responsible for collecting the various sink requests, generating the appropriate overlay network, starting, and managing the flow of data.
- *Intermediate Node*. A node in the overlay network capable of hosting some portion of the transformation operations required. An intermediate node is not restricted from residing on the same physical device(s) as the source or sink(s).

- *Controller*. On each node capable of hosting transform(s), potentially including source and sink nodes, the controller daemon understands the data traffic flowing through the node and the filters and transforms to be applied to each distinct data stream. This is also the component responsible for compiling the transformation description generated for the intermediate form into the EVPath physical representation for the node. Once the node has been configured, the controller is not involved in the data path.

3.2. Software Architecture

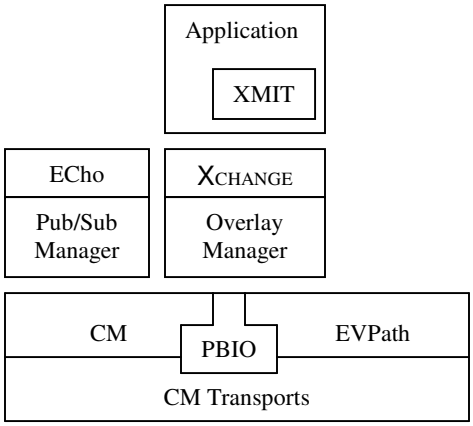

Figure 2: Software Architecture of XCHANGE

Figure 2 describes the software architecture of EVPath, ECho, and XCHANGE. The base layer uses EVPath and the PBIO [2] binary data formats, which jointly implement an efficient binary transport of typed data with automatic conversion to local platform formats. The CM (Connection Manager) [7] manages connections whereas EVPath enhances the basic functionality of CM with concrete representations for data manipulations (termed 'stones') and with the aforementioned control methods for deploying stones, linking them to other stones, and deleting them [6]. The set of 'stepping stones' crossed by formatted data events create dynamic paths across the overlay network and such 'stones' can be linked to form arbitrary graphs. At each stone, filters and translation code can be installed to alter the way in which the data being streamed is routed and manipulated. The data filters and transforms used in this paper are written in E-Code [9], but EVPath also supports other methods for dynamic code deployment. Higher-level messaging semantics, like publish/subscribe, are implemented as control methods layered on top of the EVPath infrastructure. We currently support two different sets of semantics, one being ECho's publish/subscribe, the other implementing an information flow model. The top layer consists of the end-user application and the XMIT tool [8], which can compile an XML schema [10] into a PBIO data format description and vice versa. There are three main components to the system: the controller, optimizer, and distributor.

3.3 Controller

The *controller* is realized as a daemon installed on all potentially participating overlay nodes. In addition to handling maintenance tasks, this daemon includes a code and intermediate type generator based on the data transformation descriptor provided by the application.

3.4 Optimizer

The *optimizer* is a centralized component that generates the transformation flow graph for all connected clients. The general architecture of the optimizer and its role in dynamic code generation are depicted in Figure 3.

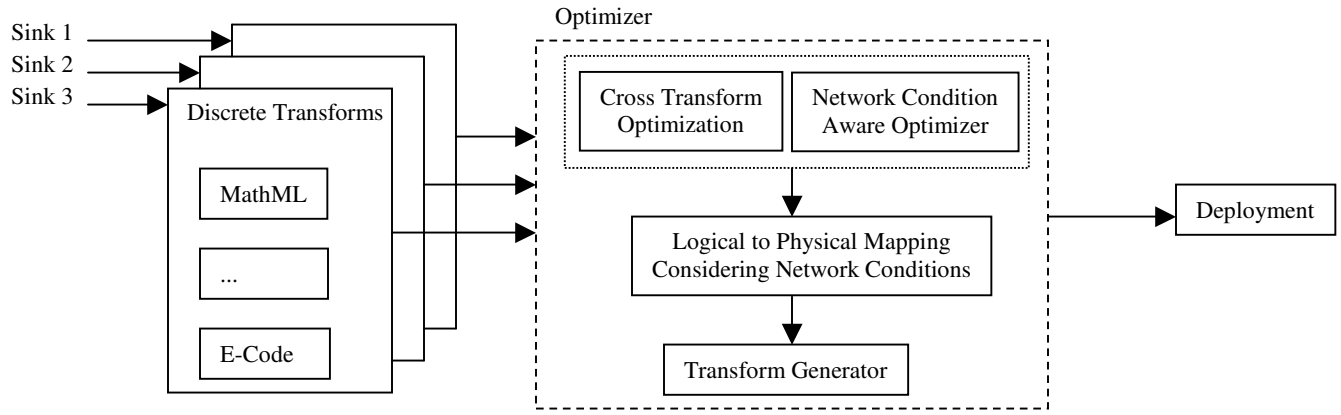

Figure 3: Overview of Data Transformation Generation Process

The operation of the optimizer is driven from the *transformation descriptor* file(s). That is, each sink provides an XML file describing the source and destination formats using an XML schema for each and a series of declarative transformation descriptions of how to adapt the source data format into the destination data format. The optimizer considers these along with the other resource constraints to determine the decomposition for deployment. Two examples of such descriptions are described next are derived from the two principal applications used in our work, the high performance remote data visualization and the enterprise health care example (see Section 5).

3.4.1 Describing Transformation Operations

The transformation generation process shown in Figure 3 uses XML-based descriptions where each transformation description is an XML node named “assign” consisting of a “dst” and a “src” element. The “dst” element describes the destination format in which the result of the transformation should be stored. This may be either a simple item or a compound item to describe a path through the type “tree” to the element. For an array or block element, this is likely just the name of the array or block rather than a particular element. The “src” element

consists of one of several items describing both the element(s) from which the destination element is formed and the operation to apply.

An example used in remote data visualization involves a 50% reduction in height and width of a 2-dimensional array of, say, a bitmap image (25% of original data size), could be achieved by using a "reduce" element listing the factor and the source element(s). Code generation at the intermediate node controller understands how to generate the proper looping and other control constructs to apply the operation to the range of elements specified. The subscript attribute of the item node represents how to apply the mapping of the elements.

```
<assign>
  <dst>
    <compound>
      <item name="pixel" subscript="x"/>
      <item name="line" subscript="y"/>
    </compound>
  </dst>
  <src>
    <reduce factor="2">
      <compound>
        <item name="pixel" subscript="x"/>
        <item name="line" subscript="y"/>
      </compound>
    </reduce>
  </src>
</assign>
```

For formulaic math operations, the MathML [11] standard is employed. This example shows a Body Mass Index (BMI) calculation, used in the healthcare example.

```
<assign>
  <dst>bmi</dst>
  <src>
    <apply><times/>
      <apply><divide/>
          <ci>weight</ci>
          <apply><times/>
            <ci>height</ci>
            <ci>height</ci>
          </apply>
      </apply>
      <cn>703</cn>
    </apply>
  </src>
</assign>
```

By default, all elements referenced in a "src" node are assumed to be from the original source data stream. As with the MathML standard, the ci tag represents an identifier. We use our type navigation syntax within that

element to navigate to the proper data member for that portion of the formula. We would also use that structure if we were to need a previously calculated destination element as part of the new calculation. In that case, we would add an attribute of the compound element called “src” to be of the value “dst”. To handle the non-compound cases, we also have a “simple” element that can host the “src” attribute as well.

3.4.2 Code Generation and Optimization

The optimizer collects all transformation descriptor files to create the composite logical graph of all requests, ordering the transformations by merging common subgraphs first and then branching out to more unique transformations. The optimization process consists of finding a path from source to sink, via the network, so as to share transforms with those already being deployed. When the transforms assigned to the logical node (starting at the root of the graph) are a subset of the new transforms being deployed, the procedure can then look for a child node to continue to overlap the existing network. Otherwise, common elements are extracted into a new ancestor node and a new branch is created for the remaining transforms.

Building the logical deployment involves distributing transforms across the various logical nodes. Type safety is maintained once the logical deployment is complete by visiting each logical node and ‘fixing up’ the incoming and outgoing data types. The current implementation uses a simple O(n) approach, where n is the number of nodes in the graph. Proper generation of source element propagation information is an important part of this process. Generally, this process collects the data elements generated in each node as it traverses down to the leaves and brings back the required source elements as it exits the child node visits. The process correctly assembles the intermediate types representing both the data elements generated so far and the source elements required by nodes downstream from the current node. The final step in the process is to map the logical graph to the physical overlay network for proper descriptor generation and deployment. The current deployment uses a simple latency metric to determine an appropriate distribution. More complex and efficient deployment algorithms have been studied elsewhere [13] and are beyond the scope of this paper.

3.5 Distributor

The *distributor* maps the transformation flow graph onto the physical network, installs the transforms and filters on the appropriate nodes, and starts or continues the flow of data from the source to the sinks. We use the infrastructure developed in [13] for this deployment work.

4. Implementation of XCHANGE -- Discussion and Evaluation

This section discusses and evaluates some of the implementation choices made for XCHANGE, positioning XCHANGE as a set of tools useful for high performance applications and scalable server systems and to identify potential bottlenecks in the XCHANGE implementation. The micro-benchmarks performed for these purposes show that, by far, the slowest part of the current implementation is deployment, which, based on the experiments in [13], grows roughly linearly in time with the number of nodes being deployed.

Using XML-described Transformations. A common criticism of using XML for high-end server applications is the relatively high cost of data parsing. XCHANGE does not experience such costs in the data fast path due to its binary representations of data and data transformations. Instead, with XCHANGE, the parsing costs experienced at runtime concern code generation, that is, we must parse the XML-described contents of transformation description files. For our XML parser, we are using the XML C Parser and Toolkit of Gnome (LibXML2) [15] in C++ with the SAX parser. Consistent with similar evaluations in [52], parsing costs for the test examples used for the scientific and healthcare applications described in this paper are not high. For both, results show that parsing overheads are less than 1 microsecond thereby demonstrating the viability of frequent runtime code generation for long-running data-driven applications.

Generating logical operator graphs. For brevity, we do not describe in detail the costs of graph generation other than commenting that overall generation cost are $O(mn^2)$, where m is the depth of the logical graph and n is the number of transforms in the client request. The actual transformations used in the applications described in this paper exhibit graph generation times of less than 1 microsecond for each case.

Mapping operator graphs to the overlay and to physical machines. Using the mapping algorithm described in [13], initial deployments of logical graphs to overlays across 10s of machines can be carried out in timeframes ranging from 200 to 800 milliseconds. Steps taken involve the transmission of XML layout descriptors to overlay controllers, parsing and compiling of XML specifications to appropriate machine code, and the assembly of the intranode data flow and transforms from the incoming stream to each of the outgoing streams. Since transforms are generated during the logical and physical mapping steps, mapping simply involves a translation operation to change the XML transform descriptions into PBIO schemas and then into ECL. The costs of creating and linking stones and deploying and having the ECL compiled for the transforms are sufficiently small that all of the test examples have a << 1 microsecond total duration.

5. Experimental Evaluation

To evaluate the efficacy of XCHANGE, two different scenarios are examined. The first scenario emulates the scientific applications implemented with SmartPointer by evaluating the costs associated with data transport (and conversion) from a single source to three sinks. The second scenario is representative of event message flows in a hospital's computing infrastructure using a single source and a single sink. Two different metrics are used. The first focuses on the network resources; the second considers processing overheads. All measurements are performed on an IBM BladeCenter with 14 HS20 blade servers installed (hostnames awing1-awing14). Each blade has dual 2.8GHz Xeon processors with 1GB RAM and a 1 Gb/s NIC card running RH Linux 9.0. Source, intermediate, and sink nodes participating are each hosted on a different blade.

5.1. Scientific Collaboration – Experiment Configuration

A specific configuration of Smartpointer is one in which the framework is used to generate visualizations of certain molecular dynamics processes for different clients. A small depiction of molecular data uses a 640 by 480 by 3 byte color image as source data. Two clients each require different image formulations. The first client has limited bandwidth and display capabilities and desires a 320 by 240 by 1 byte grayscale image where the conversion to grayscale is performed by the equation $p = (R \times 0.299 + G \times 0.587 + B \times 0.114) + 0.5$. The second does not care about color and would rather be sure there is sufficient bandwidth to handle the data load, thus requests a 640 by 480 by 1 byte grayscale image instead of the source format. The third client desires a full color, full size image.

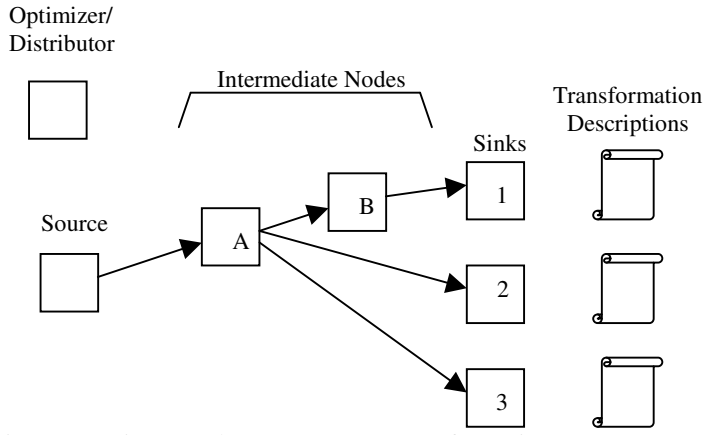

Figure 4: Using Overlays to Execute Transformations

In Figure 4, Sink 1 has its transformations split across two nodes. Sink 2 has its single transformation on one intermediate node. Sink 3 does not need any transformations and just extracts a copy of the original data stream. The optimizer recognizes the identical grayscale conversion for Sinks 1 and 2 merges the two operations. For

comparison, measurements are also taken when all of the transformations are hosted on the source and separately, when each sink hosts its own transformations.

To generate a broader understanding of the impact of data volume, we vary the example to include different sized source images. In additional tests, we manually scale all image sizes up by 50% and 100% and also down by 50% to compare the relative savings our approach offers in each of the four scenarios. To maintain a consistent comparison base, we also scale the client requests similarly for each test.

5.2 Scientific Collaboration – Experiment Results

The experiments are driven by limitations in network bandwidth due to both the large data volumes that commonly occur and the fact that an inappropriate implementation would result in the source data volume becoming the union of all potential client needs. The optimization metric considered is end-to-end delay since online collaboration is not feasible without reasonable delay constraints on the data streams being produced, transported, and displayed. Since the end node machines may not be able to deliver services to meet the needs of multiple clients, the approach taken includes offloading processing to intermediate nodes.

Measurements of the network bandwidths required for each of source, sink, and overlay hosting of the data morphing code are presented below. These measurements involve the transformation graph automatically (and dynamically) generated by XCHANGE and illustrated in Figure 5.

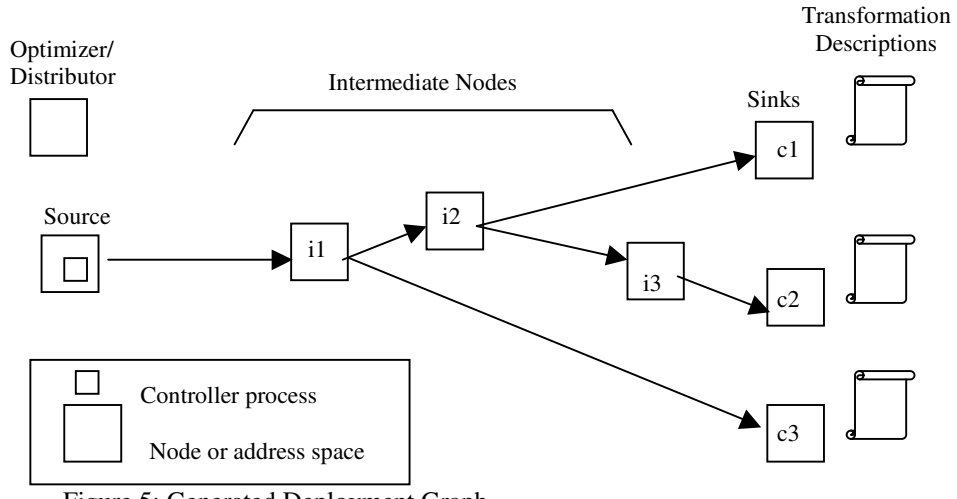

Figure 5: Generated Deployment Graph

For this set of experiments, XCHANGE is configured to only morph data on intermediate nodes, only on the source, or only on the sinks. The nodes labeled i1, i2, and i3 will have the code generated from the various

transforms installed as appropriate for the deployment model. Figure 6 displays the results. The labels correspond to the links pictured in Figure 5. The stacking of each bar corresponds to the ordering in the legend at the right.

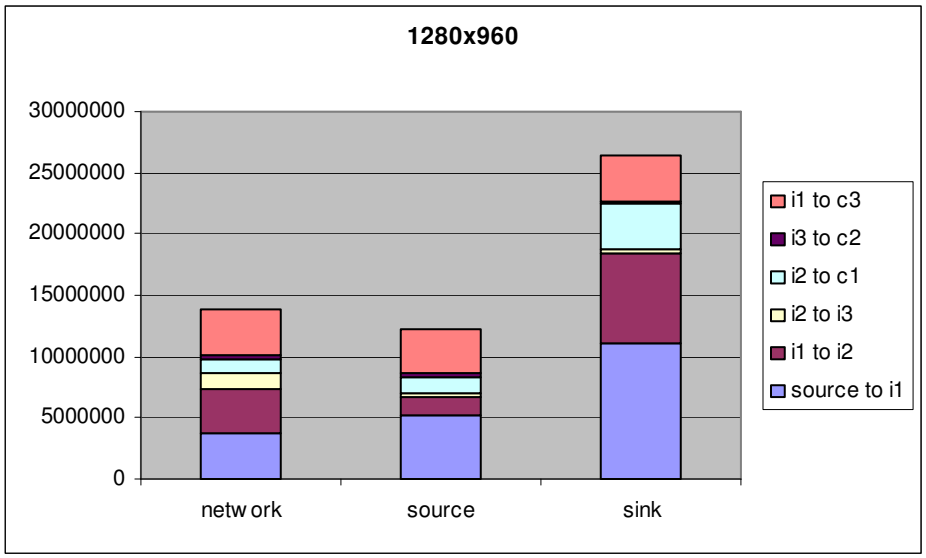

Figure 6: Scientific Example Performance Graph for a 1280x960 Image Size

The performance results presented in Figure 6 support the claim that XCHANGE can generate efficient service overlay networks. The (representative) measurements depicted in Figure 6 show that the network hosted model for dynamically generated data morphing code is nearly as bandwidth efficient overall as the source hosted model. Of particular note is that the "last mile" portion of the transmission has been offloaded from both ends, moving these potential bottlenecks into the overlay network infrastructures that generally have better bandwidths. One proposed future investigation would evaluate schemes for determining the efficacy and appropriate conditions to duplicate intermediate nodes to further reduce single link bottlenecks.

5.3 Healthcare – Experiment Setup

Consider a general hospital infrastructure consisting of a wide variety of specialized departmental systems and more general systems for cross-departmental use [35]. These systems exchange information through event messaging such as an admission event, ordering a lab test, or the emission of a set of lab results. Each departmental system maintains a private data repository tracking patient state information to guide data retention and messaging operations. Since hospitals usually buy specific systems for each department, a typical installation is comprised of a large number of different vendor products. In response, the HL-7 [22] standard for healthcare messaging codifies the

format and content of inter-system messages. For cross-platform compatibility, the format is text-based with markers to separate variable width fields.

Issues with the standard and its use include the following. First, its text format incurs parsing overheads that could be overcome with alternate binary representations. Second, each message generated by some system will almost certainly need to be translated to be understood or be usable by the receiving system, constituting another potential advantage of a binary representation. Third, since HL-7 is sufficiently flexible, occasionally ambiguous, and sometimes improperly implemented, the development of custom interfaces is typical, as evidenced by multiple companies providing them [23, 24, 25]. Finally, for all such tasks, budget limitations constrain the ways in which response times can be kept sufficiently low with rising patient data volumes.

One sample scenario consists of a typical hospital with ADT, transactional repository, order management, laboratory management, and medical imaging, among other systems. The ADT system is used to track admissions, discharges, and transfers among hospital locations for patients. Each ADT event must be copied to a transactional repository for per-patient, general use and to various departmental systems that need to know patient status or other patient attributes. For example, when a patient must have some laboratory tests performed, basic patient identification information must be available in the laboratory system to be able to link the results of the tests back to the patient records. Some patient attributes like pregnancy status may also be required in order to generate accurate results. Data generation events like ADT messages happen for each patient event or activity. Some of these include medical imaging, dietary orders, respiratory treatments, and pharmacy dispensing. By applying our system in this domain, redundant calculations across departments can be consolidated into the overlay network reducing compute load across all affected departments.

Two concrete examples are described in this paper. The first represents a physician ordering a complex lab test to generate panels of multiple results, each generated from testing a few different specimens. For instance, a complete metabolic panel consists of approximately 15 discrete values generated from separate urine and blood specimens. The lab system is configured to run a particular set of tests for each order based on the specimen type. It matches the specimen accession number to the electronically transmitted lab order and performs the required tests. The results are sent back as a list of components to the ordering system for further processing. In order for the results to be useful to the physician, they require units adjustment, new composite results generated by combining some of the discrete values, and separating them into subpanels. All of this work is performed to transform data into

something the transactional system can understand. The second example is the calculation of the Body Mass Index (BMI) [37]. While the calculation itself is not complex [38], it may need to be done across three or more departments in the hospital. The dietary department uses it to help determine the type of diet to provide to the patient [39]. Respiratory uses it as part of the calculation of respiratory functions [40]. The pharmacy uses it to determine appropriate dosages for prescribed medicines [41]. Large, multi-facility hospital systems can have 30-50 or even more individual hospitals and outpatient facilities [42, 43]. Savings can be attained by centralizing all of the IT systems for all of these facilities. In these scenarios, the frequency of the BMI calculations increases beyond a trivial amount of CPU through volume alone. By employing XCHANGE, we can pre-calculate this value once and 'on the fly' as the data is transferred to all of these departments thereby freeing the specialized departmental systems to focus on the core tasks for which they are designed. Another application of this technology is for anonymization of patient data to protect privacy by dynamically applying obfuscation [60] on demand rather than precalculation and storage.

5.4 Healthcare - Experiment Results

The general outcome of the experiments presented below is that using the XCHANGE framework can offload message transformation processing from primary systems with no negative effect on end-to-end time.

One experiment consists of the transmission of a set of 16 components where the receiving system requires components to be decomposed into 3 panels with 5 components each. Further, one of the components must have its units scaled to a range more familiar to physicians and another will be the aggregate of two components with a simple multiplicative relationship of ($Z = Y \times X$). Using three nodes, a lab system (source), an intermediate node (messaging hub), and a sink (transactional system), measurements demonstrate that there are no discernible differences in the end-to-end delay (2.00 seconds total for 10000 events) or in the volume of data hosted among any of our source, 'in network', or sink hosted models. Thus, the goal to offload computing from the server with little to no impact on end-to-end delay is attained.

The second experiment consists of calculating the BMI on an ADT message transmission. For simplicity, a typical ADT admission event is stripped down to its most relevant fields. High event volumes are emulated to evaluate potential savings in high end environments. In contrast to the lab example above, we choose to simulate the possible event load for a busy night across all hospitals in a large provider network. We consider a volume of 600 patients [44] across 50 hospitals for a total of 30000 events. To generate the events as quickly as possible, three source systems feed a single intermediate node that propagates all results to a single departmental system.

Measurements show no discernible differences in end-to-end latency when hosting the calculation on the intermediate vs. the destination node. We observe that all 30000 events are processed in 1.03 seconds when hosting it on the server. When hosting on the intermediate node or on the sink, we reduce the time to 0.85 seconds in both cases, a 17% savings. Therefore, by pulling the calculation from the server into the overlay network, the time to transmit events and the CPU load on the server are reduced.

5.5 Results Summary

The dynamic, resource-aware generation and deployment of data morphing code can improve network resource utilization and/or reduce server loads without negative effects on end-to-end latency. The scientific example demonstrates the ability of XCHANGE to reduce per-link resource costs and/or adapt to available CPU resources as appropriate for current conditions. Network hosted data morphing shows a 64% savings over sink hosted transforms and a 14% savings over source hosted transformations. The lab results healthcare example does not experience performance gains, but demonstrates the ability to offload services from the host without affecting end-to-end performance. For the BMI calculation example, by hosting morphing code in the network, we demonstrate a 17% improvement over the source hosted model.

Experiments also highlight that code generation costs are small, particularly when compared to deployment delays. That is, initial deployment performance dominates generation costs by taking 100s of milliseconds for 10s of nodes. Further, results show that it is important to dynamically generate data morphing code, as demonstrated by the scientific example, which shows high variability in network link costs for hosting generated code on the source, in the network, or on the sink. By evaluating current resource availabilities, code generation can change how morphing code is organized and then deploy the generated code differently to reduce the impact on the most constrained resource.

The current optimizations used in XCHANGE are straightforward involving simple common full expression elimination rather than the more complex common subexpression elimination. Our distributed transforms and the requirement to properly generate intermediate types add some additional complexity we have chosen not to explore at this time. Other possibilities like common subexpression elimination, operation reordering, dead code elimination, and other techniques common from the compilers field could also be employed. Additional useful optimization approaches and techniques are evaluated in [13].

An important element of this work is the ability to add new clients at runtime and merge them into a running overlay network. XCHANGE deals with this without interrupting the currently running system by first deploying the new network of nodes in parallel, setting up the full subscription model, and finally sending the data over the new overlay structure. We do not currently handle the transfer of any privately stored state in a transform function. Future extensions to this work will handle such stateful cases.

6. Conclusions and Future Work

XCHANGE shows that by performing code generation for data transforms at runtime, i.e., dynamic data morphing, and in overlay networks can lead to substantial performance improvements over static approaches. This even holds when code generation uses only rudimentary optimizations like merging common operations. This is because runtime generation can consider both current resource availability and end user needs. Experimental results attained with the Smartpointer interactive high performance application establish a 64% improvement for the 'in network' deployment of data morphing determined by XCHANGE compared with sink hosted data transformations, and a 14% improvement compared with source hosted transformations. Experiments with a representative healthcare application show that it is possible to host operations in the overlay with either no impact or a 17% improvement in the end-to-end time compared with a source hosted model.

The point of this paper is not to develop 'best' code generation or deployment schemes. Instead, the key contribution of XCHANGE is its ability to dynamically generate code from declarative descriptions of the data transforms necessary in distributed applications using the deployment schemes and performance metrics most appropriate for individual applications and systems (e.g., CPU availability, network bandwidth availability, …). Therefore, our future work will focus more on functionality than performance improvements. One idea is to use domain-specific languages to specify certain desired data semantics such as desired data resolution in a multi-scale modeling code. Another direction is to better support incremental changes to existing deployments, to add new clients to an existing deployment with small impact on the currently running application, or in response to a dynamic assessment that redeployment is necessary to continue to meet application requirements. We will also consider additional optimization techniques such as how and when to duplicate nodes that would normally be merged in order to generate a better data flow. For example, when there is a sufficiently large bottleneck for a particular node, it may be worth duplicating the transforms on another node and splitting the traffic over the multiple nodes. An area where we have done some initial work is to evaluate doing low-level common subexpression elimination rather than

performing the optimizations at the full expression level. Note that Gryphon uses a similar idea using a SQL-style data model [36].

A shortcoming of the current implementation of XCHANGE is its inability to deal with multiple similar sources serving a set of clients. This would involve determining how and when to combine operators to form a graph with disparate sources feeding into a network that ultimately connects with the various sinks. One possible approach may be to meld multiple data streams into a coherent model before considering client requests.

7. References

[1] M. Wolf, Z. Cai, W. Huang, and K. Schwan, "SmartPointers: Personalized Scientific Data Portals in your Hand," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Supercomputing '02)*: IEEE CS Press, 2002, pp. 1-16.

[2] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener, "Efficient Wire Formats for High Performance Computing", *ACM Supercomputing 2000*, (SC 2000), Nov. 2000.

[3] S. Cluet, S. Jacqmin, and J. Simeon. "The New YATL: Design and Specifications." Technical Report, INRIA, 1999.

[5] Eriksson, H., "MBONE: The Multicast Backbone", Communications of the ACM, Vol. 37, No. 8, 1994, pp. 54-60.

[6] Greg Eisenhauer, Fabian Bustamente and Karsten Schwan. "Event Services for High Performance Computing," *Proceedings of High Performance Distributed Computing (HPDC-2000).*

[7] Greg Eisenhauer. "The Connection Manager Library". August 17, 2004. Unpublished.

[8] Patrick Widener, Greg Eisenhauer, and Karsten Schwan. "Open Metadata Formats: Efficient XML-Based Communication for High Performance Computing". *Proc. Tenth IEEE International Symposium on High Performance Distributed Computing-10 (HPDC-10)*, San Francisco, California, August 6-9, 2001.

[9] Greg Eisenhauer. "Remote Application-level Processing through Derived Event Channels in ECho," Unpublished.

[10] W3C XML Schema. http://www.w3.org/XML/Schema.

[11] W3C MathML. http://www.w3.org/Math.

[12] Michael Beynon, Renato A Ferreira, Tahsin M Kurc, Alan Sussman, Joel H Saltz, "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems", *Eighth NASA Goddard*

Conference on Mass Storage Systems and Technologies/Seventeenth IEEE Symposium on Mass Storage Systems, 2000, pp. 119-133.

[13] Vibhore Kumar, Brian F Cooper, Zhongtang Cai, Greg Eisenhauer, Karsten Schwan. "Resource-Aware Distributed Stream Management using Dynamic Overlays." *25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005)*, Columbus, Ohio, USA.

[14] Weng, L.; Gagan Agrawal; Catalyurek, U.; Kur, T.; Narayanan, S.; Saltz, J. "An Approach for Automatic Data Virtualization," High Performance Distributed Computing, 2004. *Proceedings. 13th IEEE International Symposium on*, Vol., Iss., 4-6 June 2004 Pages: 24-33.

[15] XML C Parser and Toolkit. http://xmlsoft.org/.

[16] Manuel Rodríguez-Martínez, Nick Roussopoulos, "MOCHA: a Self-extensible Database Middleware System for Distributed Data Sources", *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, p.213-224, May 15-18, 2000, Dallas, Texas, United States.

[17] C. Pu, K. Schwan, and J. Walpole. "Infosphere Project: System Support for Information Flow Applications." *ACM SIGMOD Record*, 30(1), March 2001.

[18] Rodriguez-Martinez, M.; Roussopoulos, N.; McGann, J.M.; Kelley, S.; Mokwa, J.; White, B.; Jala, J.; "Integrating Distributed Scientific Data Sources with MOCHA and XRoaster." Scientific and Statistical Database Management, 2001. SSDBM 2001. *Proceedings of the Thirteenth International Conference on Distributed Data Management,* 18-20 July 2001 Pages: 263 – 266.

[19] S Babu, J Widom "Continuous Queries over Data Streams." *SIGMOD Record* 30(3):109-120 (2001)

[20] D Carney, U Cetintemel, M Cherniack, C Convey, S Lee, G Seidman, M Stonebraker, N Tatbul, S Zdonik. "Monitoring Streams: A new class of data management applications." *In Proceedings of the 27th International Conference on Very Large Databases*, Hong Kong, August 2002.

[21] Jeyarajan Thiyagalingam, Olav Beckmann and Paul H. J. Kelly. "Is Morton Layout Competitive for Large Two Dimensional Arrays, Yet?" To appear in Concurrency And Computation: Practice and Experience (special issue on Compilers for Parallel Computing, accepted February 2004).

[22] Health Level Seven. http://www.hl7.org.

[23] Healthcare Communications, Inc. Cloverleaf. http://www.healthcare.com/.

[24] SeeBeyond. e*Gate. http://www.seebeyond.com/

[25] MDI Solutions. MD Link. http://www.mdisolutions.com/

[26] Clayton, J.D. and McDowell, D.L., "A Multiscale Multiplicative Decomposition for Elastoplasticity of Polycrystals," *International Journal of Plasticity*, Vol. 19, No. 9, 2003, pp. 1401-1444.

[27] Bao, H., Bielak, J., Ghattas, O., O'Hallaron, D., Kallivokas, L., Shewchuk, J., and Xu, J. "Earthquake Ground Motion Modeling on Parallel Computers." *In Proceedings of Supercomputing '96* (Pittsburgh, PA, Nov. 1996).

[28] S. Madden and M. J. Franklin. "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data." In *ICDE Conference*, 2002.

[29] Ada Gavrilovska, Sanjay Kumar, and Karsten Schwan. "The Execution of Event-Action Rules on Programmable Network Processors." *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure* (OASIS 2004), held in conjunction with ASPLOS-XI, Boston, MA, Oct. 2004.

[30] Radhakrishnan, S. "Speed Web Delivery with HTTP Compression: A look at the Page-delivery Effects of Data Compression in HTTP 1.1". *IBM Developer Works*. 22 July 2003. http://www-106.ibm.com/developerworks/web/library/wa-httpcomp/

[31] Dong Zhou, Karsten Schwan, Greg Eisenhauer, Yuan Chen . "JECho - Interactive High Performance Computing with Java Event Channels," *in the Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS 2001)*, April 2001.

[32] Zhongtang Cai, Greg Eisenhauer, Christian Poellabauer, Karsten Schwan and Matthew Wolf. "IQ-Services: Resource-Aware Middleware for Heterogeneous Applications," *Proc. of the 13th Heterogenous Computing Workshop (HCW 2004)*, invited paper, April 2004, Santa Fe, NM.

[33] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. "Cluster-based Scalable Network Services." *In Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, Oct. 1997.

[34] Enterprise Java Beans. http://java.sun.com/products/ejb/.

[35] Hasselbring, Wilhem. "Extending the Schema Architecture of Federated Database Systems for Replicating Information in Hospital Information Systems," *Engineering Federated Database Systems EFDBS '97*, June 1997, pp 33-44.

[36] Banavar, Guruduth, et. al. “Information Flow Based Event Distribution Middleware,” *Electronic Commerce and Web-based Applications/Middleware*, 1999, pp 114-121.

[37] BMI – Body Mass Index: Introduction | DNPA | CDC. Retrieved October 15, 2005 from http://www.cdc.gov/nccdphp/dnpa/bmi/index.htm.

[38] BMI - Body Mass Index: BMI Formula for Adults | DNPA | CDC. Retrieved October 15, 2005 from http://www.cdc.gov/nccdphp/dnpa/bmi/bmi-adult-formula.htm.

[39] New Jersey State Act Enforcement Letter. Retrieved October 15, 2005 from http://www.state.nj.us/health/hcsa/hospfines/ber070104.pdf.

[40] De Lorenzo A, Petrone-De Luca P, Sasso G, Carbonelli M, Rossi P, Brancati A: “Effects of Weight Loss on Body Composition and Pulmonary Function. Respiration” 1999;66:407-412 (DOI: 10.1159/000029423) [http://content.karger.com/ProdukteDB/produkte.asp?Aktion=ShowPDF&ProduktNr=224278&Ausgabe=226555&ArtikelNr=29423&filename=29423.pdf]

[41] The Legal Practice of Pharmacy in Ohio. http://pharmacy.ohio.gov/leglprac-040901.htm

[42] Triad Hospitals, Inc. http://www.triadhospitals.com/

[43] Hospital Corporation of America (HCA). http://www.hcahealthcare.com/

[44] Private conversation with Cheryl Bronczyk, McKesson Implementation Project Manager October 24, 2005.

[45] Greg Eisenhauer, Fabian Bustamante and Karsten Schwan. “A Middleware Toolkit for Client-Initiated Service Specialization”. *Principles of Distributed Computing (PODC 2000) Middleware Symposium*, July 18-20, 2000.

[46] Wiseman Y., Schwan K. & Widener P. “Efficient End to End Data Exchange Using Configurable Compression” *Proc. The 24th IEEE Conference on Distributed Computing Systems* (ICDCS 2004), Tokyo, Japan, pp. 228-235, 2004.

[47] V.Oleson, K.Schwan, G.Eisenhauer, B.Plale, C.Pu, D.Amin. “Operational Information Systems - An Example from the Airline Industry.” *First Workshop on Industrial Experiences with Systems Software WIESS 2000*, October 2000.

[48] R. Ferraro, T. Sato, G. Brasseur, C. Deluca, and E. Guilyard, “Modeling the Earth System,” presented at *International Geoscience & Remote Sensing Symposium* (invited paper), 2003.

[49] A. Dai, A. Hu, G. A. Meehl, W. M. Washington, and W. G. Strand, "North Atlantic Ocean Circulation Changes in a Millennial Control Run and Projected Future Climates," *Journal of Climate*, (in press).

[50] M. Parashar, H. Klie, U. Catalyurek, T. Kurc, V. Matossian, J. Sltz, and M. F. Wheeler, "Application of Grid-Enable Technologies for Solving Optimization Problems in Data-Driven Reservoir Studies," presented at *4th International Conference on Computer Science* (ICCS 2004), Krakow, Poland, 2004.

[51] P. R. Pietzuch and J. M. Bacon. Hermes, "A Distributed Event-Based Middleware Architecture," in *Proc. of the 1st Int. Workshop on Distributed Event-Based Systems* (DEBS'02), pages 611-618, Vienna, Austria, July 2002.

[52] Bayardo, R. J., Gruhl, D., Josifovski, V., and Myllymaki, J., "An evaluation of binary xml encoding optimizations for fast stream based xml processing," in *Proceedings of the 13th international Conference on World Wide Web* (WWW '04), New York, NY, USA, May 17 - 20, 2004.

[53] Shetal Shah, Krithi Ramamritham, Prashant Shenoy, "Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 7, pp. 799-812, July, 2004.

[54] Ling Liu, Calton Pu, Wei Han, "XWrap: An XML-enabled Wrapper Construction System for Web Information Sources," in *Proceedinghs of the 16th International Conference on Data Engineering (ICDE 2000)*, March, 2000, San Diego, CA (IEEE CS Press). pp611-621.

[55] Ling Liu, Calton Pu, Wei Tang, Wei Han, "Conquer: A Continual Query System for Update Monitoring in the WWWW," in: special issue on Web semantics, *International journal of Computer Systems, Science and Engineering,* pp 263-304.

[56] L. Golab and M. T. Özsu, "Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams," In *Proc. 29th International Conference on Very Large Data Bases* (VLDB'03), Berlin, Germany, September 2003, pages 500-511.

[57] Butler, David, "Funneling a Flood Tide of Data: New Software Accelerates Bioinformatic Analyses," *BioPharm International*, January 2003.

[58] Sai Susarla and John Carter, "Flexible Consistency for Wide area Peer Replication," to appear in *Proceedings of the 25th International Conference on Distributed Computing Systems*, June 2005.

[59] Amol Nayate, Mike Dahlin, and Arun Iyengar, "Transparent information dissemination," in *Proceedings of the 5th ACM/IFIP/USENIX international Conference on Middleware*, Toronto, Canada, October 18 - 22, 2004.

[60] Fung, B.C.M.; Wang, K.; Yu, P.S., "Top-down specialization for information and privacy preservation," *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on* , vol., no.pp. 205- 216, 5-8 April 2005