Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

Distributed Multi-Layered Workload Synthesis for Testing Stream Processing Systems

Eric Bouillet
Parijat Dube
David George
Zhen Liu
Dimitrios Pendarakis
Li Zhang

19, Skyline Drive
IBM T. J. Watson Research Center
Hawthorne, NY 10532, U.S.A.

ABSTRACT

Testing and benchmarking of stream processing systems requires workload representative of real world scenarios with myriad of users, interacting through different applications over different modalities with different underlying protocols. The workload should have realistic volumetric and contextual statistics at different levels: user level, application level, packet level etc. Further realistic workload is inherently distributed in nature. We present a scalable framework for synthesis of distributed workload based on identifying different layers of workload corresponding to different time-scales. The architecture is extensible and modular, promotes reuse of libraries at different layers and offers the flexibility to add additional plug-ins at different layers without sacrificing the efficiency.

1 INTRODUCTION

In recent times there has been a surge in Stream Processing Systems for intelligent on-line data mining. An important aspect of Stream processing systems is the mission to receive data of all kinds, including voice, image, video, text, etc.; all at rates generally far in excess of typical clustered computer systems. Stream processing offers much to process event data that explains the importance of having proper tools for evaluating, exercising and benchmarking these systems.

Testing and benchmarking of stream processing systems requires workload representative of real world scenarios with myriad of users, interacting through different applications over different modalities with different underlying protocols. The workload should have realistic volumetric and contextual statistics at different levels: user level, application level, and packet level. Thus there is a need for generating application specific workloads with varying degree of content richness in a scalable and distributed manner.

More specifically, workload generators for testing and benchmarking stream processing systems should have the ability to simulate or emulate traffic generated by different types of applications, protocols and activities of interest to the system, such as, email, chat, web-browsing, and sensor-data (e.g. video surveillances, sensors monitoring temperature). In (Anderson et al. 2006), the authors provided a comprehensive list of requirements and challenges in developing a workload generator for testing and benchmarking distributed stream processing system dealing with high-volume, continuous, multi-modal stream data and operating in a highly resource constrained environment. The authors also reported the inadequacy of existing distributed workload generator tools like LARIAT (Rossey et al. 2002) and StreamGen (Mansour et al. 2004) in meeting these requirements.

SWORD (Scalable WORkloaD generator) is a scalable and flexible workload generator for testing and benchmarking high-volume distributed stream processing systems. It provides a distributed platform for generating a wide range of workload types with both volumetric and contextual correlations. The architecture of SWORD is shown in Figure 1. SWORD is based on a multi-agent framework supporting simultaneous creation, management and monitoring of thousands of workload generator agents in a scalable and distributed manner. The agents instantiate and invoke *Data factory Objects* which provide methods for synthesizing workload which is semantically conformant to the specific protocol. The data factory objects are implemented in C/C++ and are invoked from the agents through JNI. Data factory objects also provide functionality for generating content-rich workload by using the meta-data provided by the *Content Model*. The content model repository provides a meta-data respresentation for the semantics of the content and its statistical properties. The meta-data representation is XML based and is independent of the application specific details (modality, encoding, lower-level transport protocols). Thus different data factory objects, corresponding to different applications and protocols can access the same meta-data and independently apply application and protocol specific transformations for creating respective workloads.

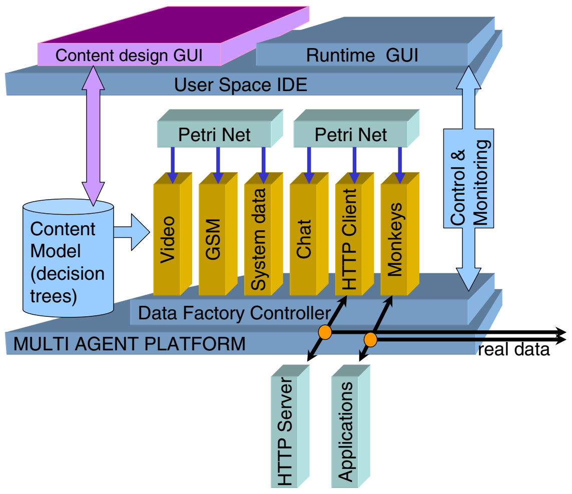

Figure 1: Architecture of SWORD.

In (Anderson et al. 2006) SWORD was introduced and its major components were overviewed from system perspectives. The contribution of this paper is to describe in detail the workload synthesis model of SWORD and report several novel features added to the model. We explain the key design choices made for different components of the workload synthesis model and how they can be exploited to support generation of statistically and semantically realistic content rich workload in a scalable, reproducible and controllable manner.

2 WORKLOAD SYNTHESIS METHODOLOGY

The programming model for workload synthesis in SWORD involves following three main components: Scenario Specification using Petri-Nets, Data Factory Objects, and Scalable Content Modeling. The design is optimized for each component without loss of flexibility. Each component is extensible and modular and the design promotes the reuse of libraries for different components: library of data factories, library of content models with meta-data that can be reused for different types of data factories, and library of (modular) Petri-Nets that can be composed into larger Petri-Nets.

2.1 Scenario Specification Using Petri-Nets

SWORD enables the generation of large numbers of streams of different types of traffic with contextual and temporal correlations. Correlations between streams can be turned on, modified or turned off dynamically; corresponding correlation parameters can take values from (random) user specified distributions. These dependencies are captured using a set of finite state machines (FSMs). FSMs describe the evolution in time of a stream, a set of dependent streams and/or the occurrence of events associated with these streams. Each FSM determines the time of instantiation, the model to be used to generate traffic and the corresponding parameters for one or more streams. Multiple FSMs can be used in parallel to generate multiple sets of correlated streams. Furthermore, FSMs can be hierarchical; a state or transition of an FSM (in the upper level of the hierarchy) leads to a new FSM (in the lower level of the hierarchy). FSMs are implemented in SWORD using colored Petri-Nets (Jensen 1997).

2.1.1 Petri-Net Model and Usage

In a typical use case scenario of SWORD, the user specifies the FSMs using colored Petri Nets.

The transitions represent events that trigger the generation of a stream (or multiple streams) using data factories (described later) with specific parameters carried in the tokens. The transitions are either immediate or delayed by an amount determined by a (random) variable. The arcs capture the dependencies between different streams; i.e., contextual correlations. The placement of tokens at initialization (*initial marking*) determines which transitions can fire and, hence, which specific content will be generated, during a round of workload generation. The parameters for each generated stream may be contained in the tokens; each token may carry different parameters, for example corresponding to a different web stream, company name or user leading to a *colored Petri-Net*.

The use of colored Petri-Nets offers the advantage of avoiding maintenance of attributes within states. Instead, attributes are assigned to tokens, which in turn maintain the state. Tokens can thus carry parameters of interest in the stream traffic generation, such as IP addresses, protocols, languages etc. An additional benefit is that to test a new scenario one just needs to modify the contents of tokens instead of modifying places. Figure 2 shows the usage of Petri-Net model for generating workload realizing insider trading activity.

2.2 Data Factories

SWORD data factories conform to a common data factory Java interface that provides the abstraction for modularity and allows extension of the data factory set with new encodings and protocols. The interface consists of initialization, process, monitoring, and finalization methods. The initialization method assigns a content model object to the data factory, and initializes the data factory from provided configuration files. It is also possible to clone data factories. The process methods executes the data factory logic

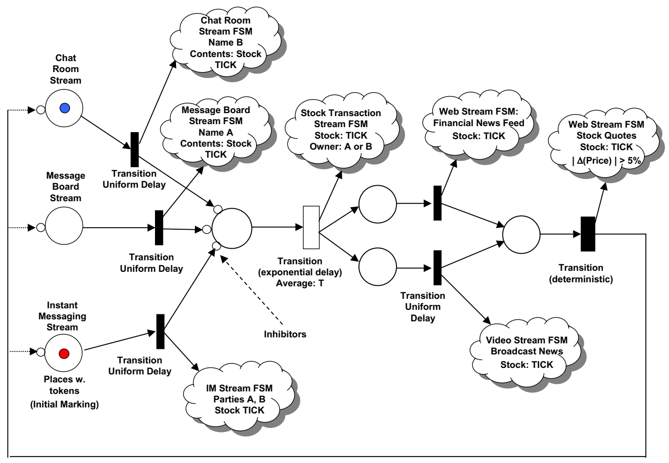

Figure 2: Example Petri Net Realization.

for transforming content meta-data (randomly) generated from the associated content model into a packet stream using specific encoding and transport protocols. The process methods typically rely on native functions to perform CPU intensive tasks, and are thus optimized in order to minimize the overhead of traversing the JNI boundary. Data factories aggregates the volumetrics of the generated content and other data factory specific statistics (such as response times for client data factories that interact with server applications). It also provides API for real time monitoring of these statistics.

High volume stream processing systems typically deal with Gb/s of traffic and hence for successfull testing and benchmarking of such systems we need to have the ability to synthesize workload of the order of Gb/s in real time. If we resort to traditional approaches of synthesizing workload as close as possible to real traces then the corresponding techniques may be quite time and resource consuming. Thus there is a tradeoff between *accuracy and scalabilty* with a higher accuracy in synthesized content having more CPU and I/O requirements and thus sacrificing scalability in content synthesis.

SWORD data factories provide the flexibility and the control mechanisms for generating workload with multiple levels of richness in the synthesized content, with varying degrees of accuracy. The different levels can be used in isolation for individual testing of different processing units or they can be mixed together in varying proportions and the resulting workload can be used for testing the aggregate system. Testing and benchmarking of different processing units can happen in parallel thus promoting modular development.

2.3 Scalable Content Modeling

In SWORD a generic data structure is created for content representation. The data structure comprises of: (1) a set of meta data content; and (2) spatial and temporal statistical dependencies among the meta data content. Within the context of SWORD, ”meta data” content is a partial, high-level representation of the content in a particular programming language. Example of meta-data content, is an XML record that contains a set of words, and a feature vector describing a human voice with Linear Prediciton Cepstrum Coefficients. This meta-data content can be fed to a text-to-speech synthesis data factory, which translates it into a waveform and streams the waveform as VoIP/RTP packets. The same set of words can be fed to a Chat data factory, which translates it into a chat message using AOL protocol, for instance.

2.3.1 Content Model Formulation

The content model is formulated in terms of *decision trees*. Nodes of the decision trees provide the logic for branching, and leafs provide the methods for generating the content meta-data which is then translated by the data factories into actual data streams. The tree pattern offers elegant extensibility and is easily learned by new users. Nodes and leafs all conform to a common set of Application Programming Interfaces (APIs) with various methods for getting the content and writing it into a buffer from where it can be accessed by data factory objects for content transformation. Nodes transparently delegate this operation to one of their children until a leaf is reached. The programmer can easily extend the decision trees with new types of decision logics and meta-data types by deriving the new implementations from this common interface.

Depending upon the type of logic implemented by a node we can classify the nodes as *Branching nodes*, *Transform nodes*, *Generator nodes* and *Cross Reference nodes*.

Branching nodes allow content selection with respect to given branching logics. Examples of branching logics are dictionaries in which branching is performed according to independent probabilities; *n*-grams allowing probabilistic content selection conditional to a history of up to *n* last branching decisions; script in which branching is executed according to a given sequence; and schedule in which branching is conditional to current simulation time; or more generally, select in which branching is dependent on the outcome of some other node.

Transform nodes provide the ability to modify the content generated by its child nodes before passing it back to its parent node. Example of transform logics are content concatenation, encapsulations (such as encapsulating into an IP header), arbitrary length repetition for generating random sentences, and logic for creating structured content. A special tranform logic generates data by providing access to push/pull ”stacks. A randomly generated content can be saved in a stack anywhere in a decision tree, and reused in other parts of the decision tree. An expression node logic allows assignment of generated content (e.g. from other decision trees) to named variables, and mathematical operations on those variables.

Generator nodes provide the mechanisms for generating integers, real values, strings, dates, and other types of arbitrary binary data. Various models have been implemented to generate constant content or random variables with prescribed distribution probabilities such as uniform, exponential or normal.

Cross Reference Nodes reference other named decision trees in the content model. If a dereferencing node is encountered during traversal of the decision tree the content generation is delegated to the decision tree referenced by the node. The referenced tree becomes a sub-tree of the decision tree. Dereferencing allows efficient sharing of meta-data content and composition of decision trees from libraries of domain-specific decision trees, such as dictionaries of words in different languages. SWORD supports local dereferencing, which is managed in-memory by the application using pointers, but also cross-application dereferencing where part of a decision tree can be generated in the decision tree of a remote application and exchanged over TCP/IP connections.

2.3.2 Dynamic Content Synthesis Architecture

The content model is stored into an XML file, and loaded into a runtime environment where it can be accessed by multiple data factories simultaneously. A content model can contain multiple decision trees, each of which is given a unique name. The binding of the data factory to a specific decision tree is done during initialization of the data factory using the name of a decision tree provided in a configuration file or in the initialization method of the data factory. The data factory uses iterators to access the designated decision tree. Iterators are a generalization of pointers with ”increment” logic. They are used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element. Each iterator maintains a local state to determine the element it will point to at the next increment, allowing multiple iterators to iterate independently over the same collection of objects. In our case the iterator iterates over an arbitrary long sequence of elements randomly generated from decision trees. Several data factories can simultaneously access the same decision tree through different iterators. Because the state information used in making decisions, including the random number generators, are maintained by the iterators, concurrent accesses are multi-thread safe without any performance penalty. Global state information that can be modified by the data factories during the decision tree traversal, such as global variables, is protected with mutexes implemented in the node logics that handle this type of information.

Whenever the data factory needs meta-data to generate the streams, it invokes its iterator’s methods to pull the content meta-data from the decision tree. The iterator starts from a node in the tree and it then follows a decision path resulting from the logics of the traversed nodes, until it reaches a leaf. Note that a complete tree traversal from root to leaf and a content generation is executed at each atomic increment of the iterator. As a consequence the iterator iterates over a sequence of meta-data content randomly generated by the leaves of the decision tree.

The iterator returns two data types into two separate buffers: *Content Meta-Data* and *Content Annotation*. Content Meta-Data is generated when the iterator reaches a leaf of the tree. The returned meta-data is used by the data factory to create the actual payloads and protocol headers of the data streams. Content Annotation is constructed while traversing the tree. It provides the information on the decision path and how the meta-data was obtained, and can be logged into a file, or sent to a different channel, where it can be used for benchmarking purpose.

In order to generate the content annotations, the user must specify a list of content annotations for the data factory’s iterator and for some or all nodes and leafs of the content model. The content annotations are of the form $< field\_name, field\_value >$. The $field\_name$ can be any string, and the $field\_value$ is a macro that returns a value. This macro conforms to the same paradigm and encoding which the decision tree used to generate the content meta-data allowing the same decision trees to be reused in the generation of both types of data. As the iterator percolates down the decision tree, all the nodes traversed by the iterator are checked for annotation fields that have the same $field\_name$ as the iterator’s annotations, and the values of the iterator’s annotations are replaced with the values of the matching nodes’ attributes.

3 DISTRIBUTED DEPLOYMENT OF SWORD

SWORD can be deployed in different manner for achieving scalability while maintaining the richness of generated content. SWORD exploits the benefits of distributed computing through the techniques of agent based architecture and distributed content generation. SWORD leverages the benefits offered by the multi-agent platform for distributed deployment of agents to achieve scalability. The content generation logic in SWORD offers the flexibility to distribute the decision trees on different system with a user-controlled coupling. The coupling can be fine-tuned in the XML content model representation to allow sharing of decision trees between different processes as and when needed. We next

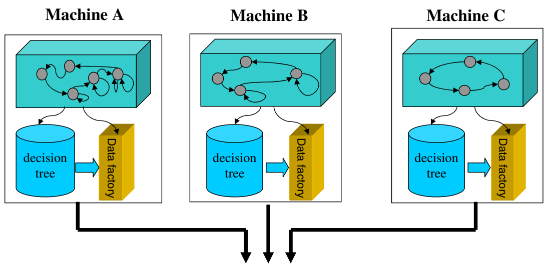

Figure 3: Distributed Deployment with No Orchestration.

present three different distributed deployments of SWORD for different different types of tests.

3.1 Distributed Deployment with No Orchestration

Figure 3 shows the deployment when different SWORD processes are running on different systems with no inter-process dependence. Though this is by far the most straightforward way of achieving scalability, this approach lacks any correlations between the workload generated by the different processes. This is because each SWORD process instance has its own scenario and content models. This type of deployment can be useful for stress testing of stream processing systems. Since the synthesized streams from different instances are independent of each other and have absolutely no correlation guarantees (either temporal or contextual) and causality, the performance benchmarks obtained by this deployment generally have the highest throughput.

3.2 Distributed Deployment with Centralised Orchestration

Figure 4 shows the second type of deployment using a central Petri-Net agent to model the scenario and a number of agents with their respective content model and data factories for generating the workload. The Petri-Net orchestrates the workload synthesis by issuing synchronous (in the same thread as the Petri-Net) or asynchronous (in a *detached* thread) calls to the agents depending upon the causality of events generated during Petri-Nets transition. The centralised coordination helps to maintain temporal and contextual correlation between workload synthesized by different agents. In this deployment, the data factory agents controlled by the Petri-Net agent do not share the states of their respective decision trees. This type of deployment can be used for testing scenarios where the decision trees for generating payload for different streams are specified apriori, e.g., the participants, application that can be invoked,

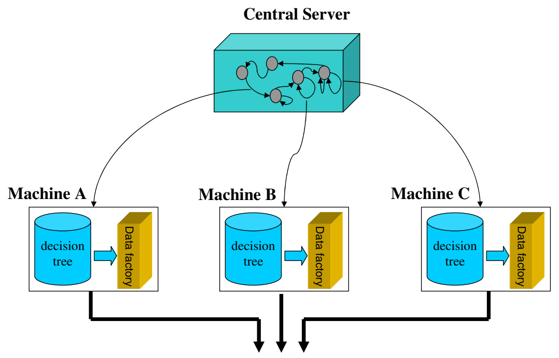

Figure 4: Distributed Deployment with Centralised Orchestration.

web pages visited by HTTP requests etc. This is typically the case when the attributes of the generated contents and hence the decision trees are static and not modifiable during run-time.

3.3 Distributed Deployment with Coupling

This deployment is similar to the previous one with the exception that the decision trees across agents can be shared using dereferencing node logics as described in Section 2.3. This allows decision trees to be dynamically augmented during run-time by linking to remote decision trees of another agent or process (on same or different machines) accessed through a TCP/IP connections. Figure 5 showcases this type of deployment. This type of deployment allows the agents to share their state information; it can also be used as a mean to distribute the CPU load of the content generation. We can envision for instance a decision tree with a load balancing decision logic at the root, which invokes sub-trees in remote processes.

3.4 Distributed Deployment with Different Levels of Orchestration

SWORD can also be deployed in a hierchical manner with different degrees of orchestration tied to the different time scales of correlation between the generated streams. Since we are synthesizing workload corresponding to different layers (application layer to the network layer) and the behavior of workload at different layers has different time scales, the generated workload should also capture this multi-time scale correlation.

As an example consider the workload corresponding to internet usage. We can model the user behavior and the resulting workload using a hierarchy of timescales:

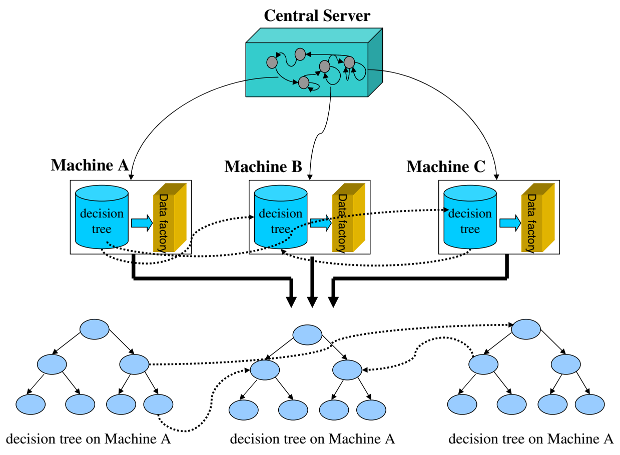
Central Server
Machine A
Machine B
Machine C
decision tree
Data factory
decision tree on Machine A
decision tree on Machine A
decision tree on Machine A

Figure 5: Distributed Deployment with Coupling.

User-level model, *Application-level model*, and *Stream level model*.

The user-level model is captured as a Petri-Net with places corresponding to time of day, application type (web-browsing, email, chat) that an individual is involved in, and transitions between different places. At this level we are not concerned with specificities (protocol level) of the particular application(s). We call the Petri-Nets modeling this level as *Level-1* Petri-Nets. The application-level model is represented as a Petri-Net with places corresponding to different possible states of an individual while using a particular application (e.g., typing, sending, clearing in case of chat), and transitions between these places. The Petri-Net at this level will generate data streams which shall constitute the generated traffic. The streams are generated in compliance with the specific protocol on which the application is running. We call the Petri-Nets modeling this level as *Level-2* Petri-Nets. Level-2 Petri-Nets represent various applications triggered by transitions in Level-1 Petri-Nets. The data generation itself is the responsibility of the data factory components. These components implement the logic for generating the content according to high-level control parameters passed-on by the application-level model, such as topic, spoken language, dictionaries, noise levels, level of realism, source (if pre-recorded). They then package this content into the appropriate stack of Protocol Data Units (PDU) before writing it to their respective output streams. Data factories can implement the complete set of operations, or simply consist of an API that emits transaction requests to real client/server settings. An example deployment is shown in Figure 6 for workload synthesis with varying levels of orchestration.

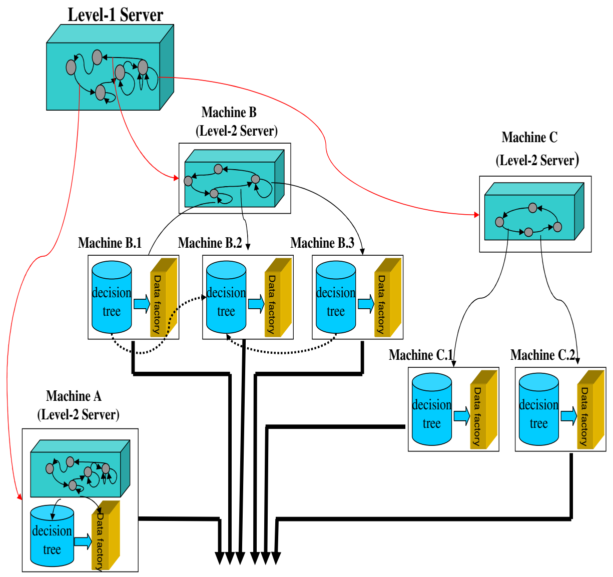

Figure 6: Distributed Deployment with Different Levels of Orchestration.

4 SWORD USAGE SCENARIOS

In this section we demonstrate the versatility of the SWORD platform with various scenarios involving a range of protocols and applications. For each scenario we describe the separation of the workload characterization into the three model components described earlier. The scenarios includes a chat session emulator; a VoIP conversation generator; and stress test generator for a data storage server. The three generators correspond respectively to the distributed deployment with no orchestration, distributed deployment with central orchestration, and distributed deployment with coupling described in Section 3.

4.1 Chat Workload Generator (Chat WLG)

The Chat WLG generates an IP packet flow with a content and a dynamic that simulates a group of persons engaged in internet chat activities. The TCP/IP packet flow simulates the packet flow captured by an IP packet sniffer located between users and chat servers. The generator can be used to validate the analytics of stream processing applications that operate on instant messaging content. Example of target applications are applications that cluster chat conversations and participants based on feature vectors collected from their chat activities. Such applications can be used to map social networks, or to profile users (e.g. determine their real age group and gender) and their possible intents. They can perform temporal analysis of the chat dynamic (time spent entering a sentence, reflection times), semantic analysis of its content (topics of interest, vocabulary, grammar), as well as analysis of the meta-data (professed user profile, buddy list, IP addresses), and it is thus important that these characteristics be accurately modeled.

We use a Petri-Net model to represent the user-level activities of a participant during a chat session as shown in Figure 7. Participants are initially "Idle". When they start typing they enter the "Typing" state, and remain in this state until they either send the message they are currently typing, or until they completely clear their message prompt, at which point they return to the "Idle" state. Transitions between those two states generate notification events to inform the participants of each other's status (i.e. whether a person is responding to a message for instance). This is illustrated as Petri-Net (1) in Figure 7. Note that this Petri-Net falls short of modeling the dynamics of a realistic chat session. In particular it represents chat participants independently of each other and as such it fails to capture behavioral correlations across the participants. Petri-Net (2) in Figure 7 addresses this problem with a more accurate model that compounds the states of two chat participants (A and B). This model provides us with a finer granularity in the specifications of the statistical properties of the transition, such as participants being more likely to wait while the other is responding. If desired, the Petri-Net can be arbitrarily extended, to express login, buddy list transfer and logout actions. In addition, it can include states to express the participant mood, topic of discussion, and even simulate participants engaging in other activities.

Petri-Net models can be packaged into libraries offering a spectrum of such user behaviors. The tradeoff of using larger Petri-Net is an increased processing overhead and footprint. As it grows larger and more complex, it also requires a thorough workload characterization in order to determine the appropriate values for the parameters. In our experiments we assumed Petri-Net (2) and we collected the corresponding temporal statistics using tcp dumps of real chat conversations. For each participant, we also conducted a least squares regression analysis between the average time T spent typing a sentence (time between the first *type* notification to the next *send* notification) and the average packet lengths W.

Each of the transition in the Petri-Net invokes a corresponding method in the data factory with the token attributes as argument. The data factory interprets the command and generate the appropriate packet. If the command requires a content (such as the text message of a *send* command), the data factory gets the randomly generated content from a provided decision tree as described later. The data factory of our experiment emulates the Sametime (TM) protocol stack down to the Ethernet layer. If needed additional data factories can be developed to handle other chat transport protocols. For the sake of simplicity, we implemented a canonical TCP protocol layer where all packets are received in-sequence without packet loss. The protocol emulation gives us some control on certain attributes that are normally beyond the control of the user, such as MAC address, IP addresses, and packet time to live.

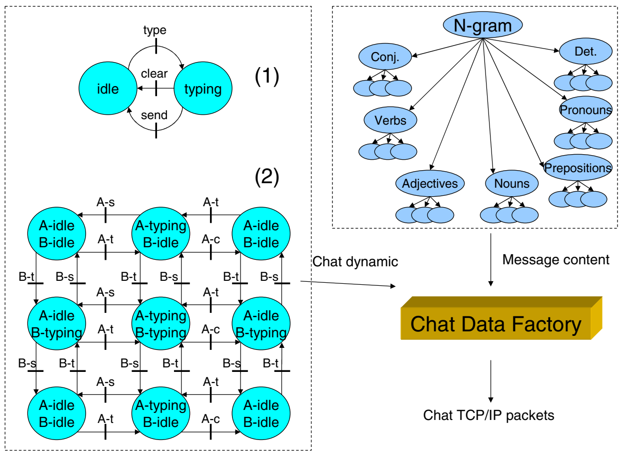

Figure 7: Workload Characterization for Chat Generator

The statistical properties of the text message are specified in the form of decision tree. At the root of the decision tree we use an N-gram logic to express the conditional probabilities of generating random sequences of word classes, and for each word-class the probability of occurrences of words within that class. In our experiments, we used a dictionary of the $10,000$ most frequent english words. The N-gram model was trained with a corpus of real chat log archives.

Using this workload characterization model it is possible to provide a core library of decision trees for english words and for other languages. It is also possible to extend those libraries with semantic domain specific content, such as dictionaries for sport related content, computer science, etc. Furthermore, libraries of dictionaries can be created to represent the word frequencies for different classes of individuals, and similarly a collection of N-gram models can enumerate the various ways of associating those words into a sentence depending on the individual's background. Through references and reuse, smaller libraries can be composed into larger ones to represent a very rich set of individuals, each marked by particular statistical characteristics.

4.2 VoIP Workload Generator (VoIP WLG)

We developed the VoIP WLG scenario for testing a customer service call center quality monitoring application.

The WLG is capable of generating hundreds of simultaneous streams of real VoIP traffic from a pre-recorded corpus of 4472 phone conversions between 679 participants per CPU. The conversations are made up of two channels each, corresponding to both directions of the conversations. The VoIP scenario is illustrated in Figure 8. We use a Petri-Net to model the dynamics of phone call arrivals between pairs of individuals. The Petri-Net consists of one place with one colored token for each participant, and one transition for each conversation between two individuals (we use a single transition to represent all conversations between the same pair of individuals). The transitions have guard clauses requiring both individuals to be available (i.e. not being involved in other transitions) in order for the transition to be allowed. Transitions have exponentially distributed wait times. When a transition is fired, a VoIP data factory is invoked in a separate thread with a pre-recorded conversation obtained from a given decision tree containing all conversations between the involved individuals. The tokens are locked during the length of the conversation and released when the conversation ends, ensuring that the same token (person) cannot be involved into two or more simultaneous conversations.

We use annotations in the decision tree to indicate for each channel, the speaker IDs, the conversation ID, the encoding, whether the speaker is caller or callee, speaker gender and age, source and destination IP addresses, and a description of the conversation which can be used to verify the quality of the call center monitoring analytics. The annotation adds an overhead of about 500 bytes on the average to every 111 bytes of the VoIP RTP packet-stream. The VoIP data factory has the ability to synchronize both channels of the conversation and send them to different IP addresses (the addresses of two IP packet sniffers located in different parts of the call center's network). In addition it is possible to configure a packet-loss ratio, and delays for each channel.

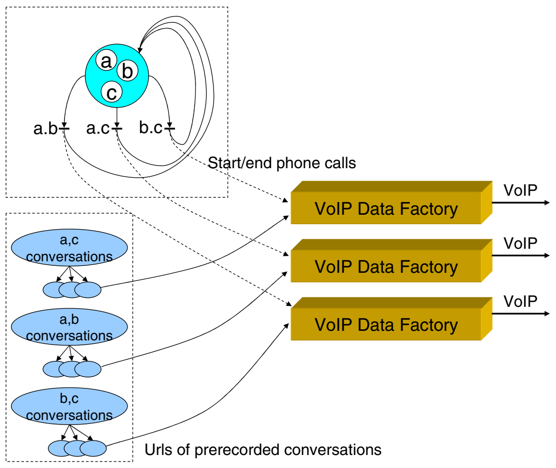

Figure 8: VoIP Workload Generator

4.3 Storage Management Stress Test Generator

The Storage Management Stress Test generator is used to test the performance of a value-based storage management which was developed for a stream processing system. In this scenario each data factory generates a sequence of WRITE, READ, or DELETE storage commands. It also generate UPDATE commands to modify the relative importance of stored data blocks. In addition the WLG can mix invalid commands in the sequence in order to test the robustness of the storage system. The Storage scenario is illustrated in Figure 9.

When generating a WRITE command, the stored data consists of arbitrary content of random length, ranging from a few bytes to several hundreds of megabytes, and a corresponding retention value. After a WRITE command the generator expects either a token ID that can be used to retrieve the stored data if the command is successful, or an error code. If a token is returned, it is inserted in a *pool*. In our content model representation a pool is a container of arbitrary data. Some of the decision tree node logics that operate on pool objects are: (1) a *Push* logic that saves the content generated by a child node into a named pool, (2) a *Pick* logic that returns a randomly selected content from a named pool, and (3) a *Pop* logic that returns a randomly selected content from a named pool and delete it from the pool. Using the pool decision tree logic it is thus possible to retrieve previously saved tokens returned with each WRITE command. The tokens can then be combined in READ and UPDATE commands using the *Pick* logic, and DELETE commands using the *Pop* logic. More importantly, since the pools are part of the decision tree representation, it can be linked to remote decision trees and shared across processes, allowing tokens resulting from WRITE commands in one process to be used in READ, UPDATE and DELETE commands generated in a separate process.

5 CONCLUSION

We have described a system for testing and benchmarking stream processing systems. We focused on the architecture, identifying salient aspects for extensibility and comprehensive multi-modal content synthesis. The described system has been applied to the needs for verifying correct stream analytics and performance testing, however, not just simply in single instances of the underlying framework (Wu et al. 2007).

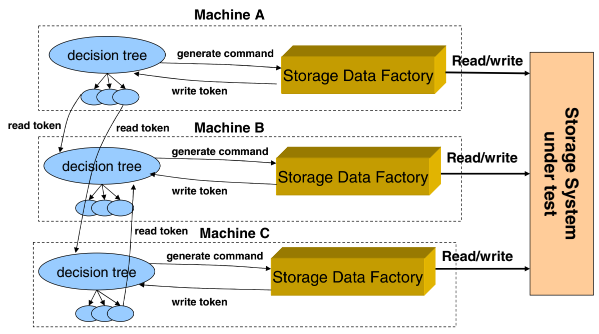

Figure 9: Storage Management Stress Test Workload Generator

REFERENCES

Anderson, K., J. P. Bigus, E. Bouillet, P. Dube, N. Halim, Z. Liu, and D. Pendarakis. 2006. Sword: Scalable and flexible workload generator for distributed stream processing systems. In *Proc. of 2006 Winter Simulation Conference*, ed. L. F. Perrone, B. G. Lawson, J. Liu, and F. P. Wieland, 2109–2116. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Jensen, K. 1997. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag.

Mansour, M., M. Wolf, and K. Schwan. 2004. Streamgen: A workload generation tool for distributed information flow applications. In *Proc. of International Conference on Parallel Processing (ICPP'04)*, 55–62.

Rossey, L. M., R. K. Cunningham, D. J. Fried, J. C. Rabek, R. P. Lippmann, J. W. Haines, and M. A. Zissman. 2002. Lariat: Lincoln adaptable real-time information assurance testbed. In *Proc. of IEEE Aerospace Conference*, 2671–2682.

Wu, K.-L., P. S. Yu, B. Gedik, K. Hildrum, C. Aggarwal, E. Bouillet, W. Fan, D. George, X. Gu, G. Luo, and H. Wang. 2007. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. In *Proc. of Very Large Database Systems (VLDB)*, 1185–1196.

AUTHOR BIOGRAPHIES

ERIC BOUILLET is currently at IBM T. J. Watson Research Center, NY, where he works on data modeling and test data generation. Before joining IBM, Dr. Bouillet has worked at Tellium Inc. from 2000-2004 and at Bell Labs/Lucent Technologies from 1998-2000. Eric holds an M.S. and a Ph.D. in electrical engineering from Columbia University. His current research interests include data modeling and test data generation, design of optical networks and optimization of lightpath provisioning and fault restoration algorithms. *Email: ericbou@us.ibm.com*.

PARIJAT DUBE received his M.S. in Electrical Communication Engg. from Indian Institute of Science, Bangalore in 2001 and his Ph.D. in Computer Science from University of Nice-Sophia Antipolis in 2002 where he was affiliated to INRIA. He joined IBM T. J. Watson Research Center, Hawthorne, New York in 2002. His research interests include performance analysis and control of computer systems, distributed computing, stochastic modeling and game theory. *Email: pdube@us.ibm.com*.

DAVID GEORGE is a Research Staff Member at the IBM T. J. Watson Research Lab. In his long career, he has researched, defined and architected new paradigms for computation. These include pipeline numerical processing, telecommunications and networking, dataflow and parallel computation, fault-tolerant systems, advanced storage concepts, testability, stream processing and policy management. He has an MSEE from Syracuse University, and a BEE from The Ohio State University.*Email: dag@us.ibm.com*.

ZHEN LIU has Ph.D in Computer Science from the University of Paris XI, France. He was with the France Telecom R&D, then joined INRIA. He is currently the senior manager of the Next Generation Distributed Systems Department at IBM T. J. Watson Research Center. Zhen Liu is a fellow of IEEE and a master inventor of IBM. He was the program co-chair of the Joint Conference of ACM Sigmetrics and IFIP Performance 2004, and the general chair of ACM Sigmetrics 2008. *Email: zhenl@us.ibm.com*.

DIMITRIOS PENDARAKIS manages the Secure Systems group at the IBM T.J. Watson Research Center. His current research interests include secure virtualization and cloud computing, trusted computing and secure hardware. Dimitrios joined IBM in 1995 after receiving his PhD from Columbia University. Between 2000 and 2003 he was Principal architect with Tellium, where he led the development of next generation control systems for intelligent optical networks. *Email: dimitris@us.ibm.com*.

LI ZHANG graduated from IEOR Dept. Columbia University after receiving degrees from Purdue University and Beijing University. He is the manager of the System Analysis and Optimization group at IBM T.J. Watson Research Center. His research interests include control and performance analysis of computer systems, statistical techniques for traffic modeling and prediction, scheduling and resource allocation in parallel and distributed systems, as well as measurement based clock synchronization mechanisms. *Email: zhangli@us.ibm.com*.