BFSiena: a Communication Substrate for StreamMine

Zbigniew Jerzak
Dresden University of Technology
Systems Engineering Group
D-01062 Dresden, Germany
Zbigniew.Jerzak@tu-dresden.de

Christof Fetzer
Dresden University of Technology
Systems Engineering Group
D-01062 Dresden, Germany
Christof.Fetzer@tu-dresden.de

ABSTRACT

StreamMine is a scalable middleware for massive real-time data streaming. In this paper we present the BF*Siena*: a communication substrate for the StreamMine. BF*Siena* is a content-based, publish/subscribe communication system which provides support for arbitrary predicate based messaging in the acyclic peer to peer networks. BF*Siena* is a low latency, high throughput communication system which is well suited for the processing frameworks, like StreamMine.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

General Terms

Algorithms, Design

Keywords

distribution, interaction, publish/subscribe, event processing

1. INTRODUCTION

StreamMine is a part of the Scalable Automatic Streaming Middleware for Real-Time Processing of Massive Data Flows (STREAM) seventh framework program (FP7-216181). The goal of STREAM is to develop a middleware which supports scalable processing of real-time streaming data. STREAM seeks to provide support for the real-time data mining applications with an intuitive, high-level query language. One of the important parts of the STREAM project is the communication framework. The design goal for the communication framework is to support flexibility by not enforcing explicit declaration of connections between components and simultaneously offering expressiveness in form of the content-based filters and dynamic information routing.

Throughout the rest of this paper we will use terms *publisher* and *subscriber* to denote the source and the sink of the *events*, respectively. Events are delivered to subscribers by a network of *brokers* which perform a *content-based* matching of *subscriptions* with events. Subscriptions consists of conjunction of *predicates*, which during the matching process are evaluated over the the disjunction of predicates stored in the events. Subscriptions are flooded into the network so as to ensure their rendezvous with the events. Alternatively, publishers might broadcast *advertisements* which summarize the set of events published and control the propagation of subscriptions. For a more comprehensive overview of the existing publish/subscribe systems and terminology we would like to refer the reader to [2].

2. DESIGN AND IMPLEMENTATION

The BF*Siena* is based on the X*Siena* research project (available for download at: http://wwwse.inf.tu-dresden.de/xsiena/) carried out by the Systems Engineering Group at the Dresden University of Technology. The X*Siena* project stems from the *Siena* content-based publish/subscribe system [1] created by Antonio Carzaniga. In the following sections we will present the design and the implementation details of the X*Siena* and the BF*Siena* projects.

2.1 X*Siena*

X*Siena* is a content-based publish/subscribe system. It uses an advertisement driven model in that events not covered by advertisements will not be delivered to subscribers. Although X*Siena* shares similarities with its protagonist, the *Siena* system, it has also a few major differences. First one is the already mentioned advertisement based routing. Second is the use of the acyclic peer-to-peer architecture. Third is a new pluggable communication infrastructure, which is currently based on the Apache Mina network application framework [1]. Finally, in order to meet the stringent requirements with respect to real-time X*Siena* implements the Fail-Aware Publish/Subscribe model presented in [3]. The routing of events, subscriptions and advertisements in the X*Siena* framework is based on the algorithms presented in [7].

Listing 1 shows the interface of the X*Siena* broker (`IXSiena`) and the X*Siena* application/subscriber (`INotifiable`). The interface design of the X*Siena* broker follows the lines of the *Siena* approach and the recommendations presented in [5].

[1] http://mina.apache.org/

```
//the broker interface
public interface IXSiena {
  void publish(CEvent n);
  void subscribe(CFilter f, INotifiable n);
  void unsubscribe(CFilter f, INotifiable n);
  void advertise(final CFilter f);
  void unadvertise(final CFilter f);
  void shutdown();
  CXSienaSocketAddress address();
}

//the application interface
public interface INotifiable {
  void notify(CEvent n);
  void cancel(CFilter f);
  CXSienaSocketAddress address();
}
```

Listing 1: The X*Siena* interface.

The only difference to [5] is that X*Siena* brokers are designed using a hard-state and not soft-state approach. The publication of advertisements (`CFilter`) and events (`CEvent`) is performed by calling the `publish()` and `advertise()` methods of the class implementing the `IXSiena` interface. The `address()` method returns the X*Siena* socket address on which the object implementing the `IXSiena` interface can be reached. The `INotifiable` interface is implemented by the applications/subscribers. It provides an upcall for: (1) delivery of events (`notify()`) and (2) for informing about previously issued subscriptions (`CFilter`) which did not match any of the advertisements published so far (`cancel()`).

Listing 2 presents a sample X*Siena* application, which has been used to generate one of the latency graphs (Figure 2(a)) in Section 3. The application creates a network consisting of three components: publisher, subscriber and a broker (`CDisptacher`), running on the one physical host and communicating via TCP protocol (see 1 broker setup in Figure 1). Publisher and subscriber use a `CThinClient` which is a gateway allowing remote hosts to connect to existing X*Siena* brokers. One `CThinClient` can host multiple applications (`CLatencyApp`) which are differentiated by their id (`C1A.setId("app1")`). The publisher first advertises its events (`P1A.advertise(f)`). Subsequently, the application subscribes and registers itself with the subscriber (`S1A.subscribe(f, LT)`) in order to be informed about events matching the subscription it has issued.

2.2 BF*Siena*

The development of the BF*Siena* was motivated by the high requirement of the StreamMine project with respect to the routing efficiency of the underlying communication infrastructure. While offering an improvement over the *Siena* routing (especially with respect to supported architectures and advertisements) X*Siena* was still showing a room for improvement. Moreover, advertisement based routing has proved to result in a too tight coupling between the publishers and subscribers. In current version, the publishers in the StreamMine system need to be able to dynamically change the type of published events and subscribers should be able to react quickly to the changing environment, without the need to wait for the advertisement propagation.

Therefore, for the development of the BF*Siena* we have used

```
//the latency measuring subscriber
class CLatencyApp implements INotifiable {
  CXSienaSocketAddress sa = null; int c;

  CLatencyApp(CXSienaSocketAddress s) {
    sa = s; c = 0;
  }
  //how can this application be reached
  public CXSienaSocketAddress address() {
    return sa;
  }
  //ignore subs with no matching advertisements
  public void cancel(CFilter f) {}
  //deliver publication to subscriber
  public void notify(CEvent e) {
    c++;
    long RT = System.nanoTime();
    long ST = e.getAttribute("ST").longValue();
    System.out.println(c+"\t"+(RT-ST)/1000));
  }
}

//the test program for latency measurements
public static void main(String[] args){
  CEvent e = new CEvent();
  e.putAttribute("test", 10);
  CFilter f = new CFilter();
  f.addConstraint("test", COp.GT, 0);
  //initialize the addresses of components
  String h = "xsiena://localhost";
  CXSienaSocketAddress B1A, P1A, S1A, LTA;
  B1A = CXSienaURI.parse(h+":4050");
  P1A = CXSienaURI.parse(h+":3030");
  S1A = CXSienaURI.parse(h+":3040");
  LTA = CXSienaURI.parse(h+":3040");
  LTA.setId("app1");
  //initialize a standalone broker...
  CDisptacher B1 = new CDispatcher(B1A, null);
  //and connect publisher and subscriber to it
  CThinClient P1 = new CThinClient(P1A, B1A);
  CThinClient S1 = new CThinClient(S1A, B1A);
  CLatencyApp LT = new CLatencyApp(LTA);

  P1A.advertise(f); //advertise events
  Thread.sleep(100);
  S1A.subscribe(f, LT); //subscribe
  Thread.sleep(100);
  ev.putAttribute("ST", System.nanoTime());
  P1A.publish(e); //publish event
}
```

Listing 2: The X*Siena* latency measurement application.

a subscription-based routing branch of the X*Siena* project. The novelty of the BF*Siena* design lies in the Bloom filters which excel the routing process and allow us to achieve high throughput without the need to scarify the expressiveness of the content-based scheme. Moreover, BF*Siena* supports a dual routing strategy which allows to select between the so called edge routing and normal routing where an event has to be matched with the whole set of subscriptions in every broker on its way from publisher to subscriber. When using the edge routing an event is matched only once in the first router it encounters in the network. The first router attaches a Bloom filter to the event. All subsequent routers use the attached Bloom filter (instead of the content of the event) to forward it to the interested subscribers. The trade-off of the edge routing approach is the need to propagate all subscriptions (except for the identical ones) to all brokers in the system. The most important aspect of the Bloom filter based routing is the fact that the developed algorithms (unlike, e.g., [8]) do not limit the expressiveness (subscription

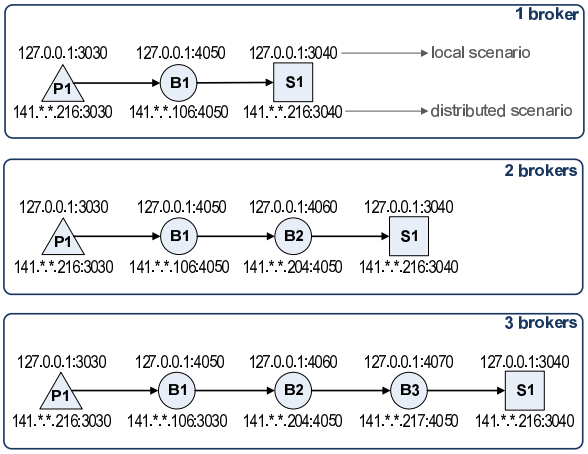


Figure 1: The experiment setup

and publication language) of the content-based publish/subscribe systems. Internally, the development of the BF*Siena* involved the replacement of the `forest`-based routing tables (see [7] for details) of the X*Siena* with two new structures: the `bfposet` and the `sbstree`. The `bfposet` stores the predicates of the subscriptions. The `sbstree` stores a disjunction of conjunctions of encoded subscription predicates. An event incoming at the edge broker is matched based on its content with the `bfposet`. The result is a Bloom filter which encodes the matching predicates. This Bloom filter is subsequently passed to the `sbstree` which computes the disjunction of conjunctions over the encoded subscriptions predicates and returns the set of interfaces to forward the event on. Subsequent brokers encountered by the event with the attached Bloom filter only evaluate the Bloom filter using the `sbstree`. Otherwise, if no edge routing is being used the matching in the `bfposet` has to be repeated. For the detailed description of the BF*Siena* routing algorithms please refer to [4].

3. EVALUATION

While evaluating the implementation of the BF*Siena* we have focused on two parameters which have the highest priority for the StreamMine project. These parameters are: (1) the latency and (2) the throughput of the BF*Siena* communication substrate. The evaluation has been performed in the distributed and local environment (see Figure 1). In both cases communication was carried out over sockets. In the local environment we have used the loopback address of one machine, whereas in the distributed scenario we have used different physical computers.

Every test, unless otherwise stated was executed in the same way. First a broker network, composed of varying number of brokers ($B1$, $B2$, $B3$) was set up. Subsequently the subscriber $S1$ issued a single subscription which was propagated through the whole network. Finally publisher $P1$ was issuing events (the flow of events is depicted by arrows in the Figure 1) which were always received by the subscriber $S1$, i.e., they always matched the subscription issued by the $S1$. An event consisted of ten different attribute name and value pairs, with the total wire size of approximately 740 bytes.

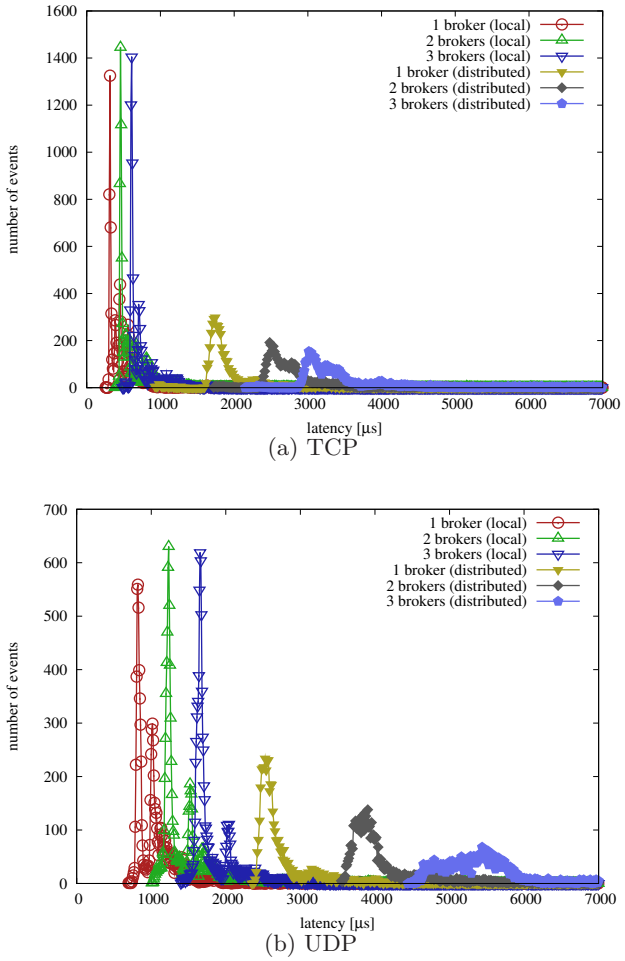
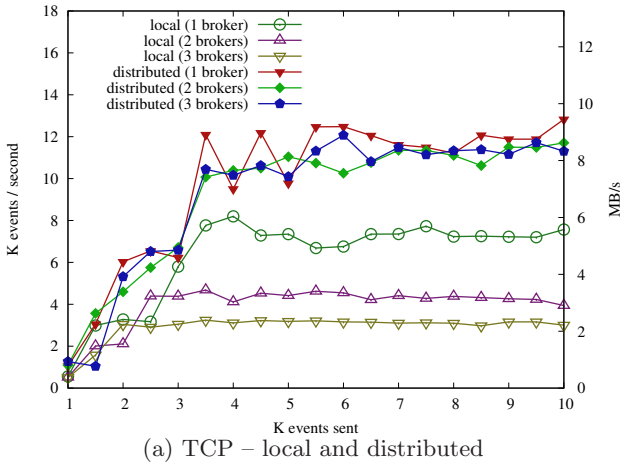


Figure 2: Latency measurements in local and distributed scenario

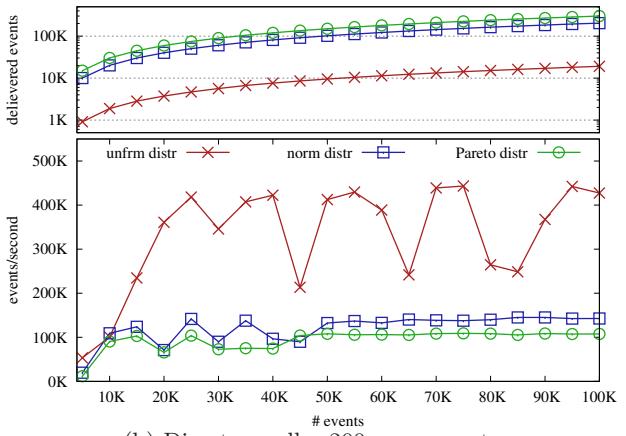
All experiments have been executed on the same hardware and software: Java HotSpot(TM) Server VM (build 1.6.0_03-b05, mixed mode) running on a DELL Optiplex 745 with Core 2 Duo E6400 processor and 2GB of RAM (publisher and subscriber) and on a DELL Optiplex 755 with Core 2 Quad Q6600 processor and 4GB of RAM (brokers). All computers are connected with a standard 100BASE-T (full duplex) Ethernet.

Figure 2 compares the latencies of the TCP and UDP based communication for different number of brokers. For the latency measurement, publisher attached a send time stamp (using the `System.nanoTime()` method) to every event it has published. The subscriber, upon reception of the event calculated the reception time stamp (`System.nanoTime()`) and logged the difference between the two (see Listing 2). Obviously, for the above scenario to work, both publisher and subscriber had to be placed on the same physical machine (`141.*.*.216`).

Both graphs (Figures 2(a) and 2(b)) for the ease of comparison share the same scale along the x axis. We can observe that for the local scenario, even with three brokers placed



(a) TCP – local and distributed



(b) Direct upcall – 200 components

Figure 3: Throughput measurements

between the publisher and the subscriber, the majority of the latencies stays in the sub-millisecond range. The connectionless UDP protocol, included for comparison reasons, exposes approximately double the latencies of the TCP and tends to be more heavy tailed. We believe that the exposed latencies are a good starting point towards meeting of the requirements of the modern real-time stream processing [6].

The second important parameter which had to be considered when planning the use of the BF*Siena* in the StreamMine project was the ability to provide high throughput for the events. Figure 3 presents the throughput tests for two scenarios: (1) communication via sockets with all brokers running on the same physical host (local) and all brokers running on different hosts (distributed) – Figure 3(a) and (2) communication via direct upcall – Figure 3(b). In case of the socket based communication (Figure 3(a)) we can observe that the throughput of the distributed runs approaches the network saturation values (100BASE-T full duplex Ethernet). For the local setup, the overhead of running the multiple brokers on the same host results in the observable throughput decrease.

The direct upcall test simulated two hundred different components of the StreamMine architecture registering their subscriptions with the BF*Siena* broker. We have performed three tests with varying number of events matching the registered filters, according to one of the distributions: (1) uniform (least overlap), (2) normal (medium overlap) and (3) Pareto (highest overlap). The type of the distribution chosen strongly influences the total number of the delivered events. The 300000 of delivered events in case of the Pareto distribution indicate that on average a single event was delivered to three components. It can be observed that the achieved throughput in all cases should satisfy even very demanding applications. The sudden drops in the throughput for the uniform distribution are the result of the Java garbage collection.

4. SUMMARY

We have presented the BF*Siena*, a communication substrate for the StreamMine project. BF*Siena* is based on X*Siena* project and is characterized by a novel Bloom filter based routing algorithm and easy to use API which allows for rapid development of applications taking advantage of the underlying content-based publish/subscribe communication infrastructure. We believe that thanks to its flexibility and performance BF*Siena* it will provide a good middleware for the StreamMine.

5. REFERENCES

[1] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[2] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[3] Zbigniew Jerzak, Robert Fach, and Christof Fetzer. Fail-aware publish/subscribe. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)*, pages 113–125, Cambridge, MA, USA, July 2007. IEEE Computer Society.

[4] Zbigniew Jerzak and Christof Fetzer. Bloom filter based routing for content-based publish/subscribe. In *Proceedings of the Second Conference on Distributed Event-Based Systems*, Rome, Italy, July 2008.

[5] Peter Pietzuch, David Eyers, Samuel Kounev, and Brian Shand. Towards a common api for publish/subscribe. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 152–157, New York, NY, USA, 2007. ACM.

[6] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[7] Sasu Tarkoma and Jaakko Kangasharju. Optimizing content-based routers: posets and forests. *Distributed Computing*, 19(1):62–77, September 2006.

[8] P. Triantafillou and A. Economides. Subscription summarization: a new paradigm for efficient publish/subscribesystems. In *Proceedings of 24th International Conference on Distributed Computing Systems*, 2004.