Hierarchical Dissemination of Streams in Content Distribution Networks

Shrideep Pallickara and Geoffrey Fox
Community Grids Lab, Indiana University

1 Introduction

Streaming pertains to the routing of data streams from the sources to those entities that are interested in them. In streaming, the dissemination of streams is typically independent of the underlying network and is, instead, content-based. The routing is within the purview of the content distribution network which tracks both the entities and their interests. Content distribution networks provide a scalable framework for exchanging information between a very large number of entities. These content distribution networks could be based on multicast, peer-t-peer, publish/subscribe or ad hoc networking. This work focuses on the hierarchical dissemination of streams in content distribution networks based on publish/subscribe.

By decoupling the roles of producers and consumers of a data stream, publish/subscribe systems provide a loosely-coupled framework for streaming. Producers of data streams include metadata describing the content encapsulated in a given stream fragment. These content descriptors are referred to as topics. Consumers specify their interests in consuming portions of a stream through *subscriptions* – constraints specified on the values that the content descriptors might take. The complexity of a subscription is directly proportional to the richness of the content description. The content distribution network disseminates streams based on the registered subscriptions and the stream's content descriptors.

The simplest content descriptor is a String, for e.g. Sensor/Streams. This simplicity also enables extremely fast evaluations of whether a stream fragment satisfies a specified subscription constraint. *Hierarchical content description* assumes that the "/" in the content descriptors are significant, and correspond to finer-grained descriptions. Thus, Streams/Sensor/Fluid would describe streams produced by all sensors reporting on various fluid properties, while Streams/Sensor/Fluid/Pressure would describe streams produced by a Piezometer, which is used to measure fluid pressure.

Hierarchical streaming simplifies the process of registering interest in content. Without support for hierarchical streaming, every consumer would need to be aware of every finer-grained description of content. The case for hierarchical streaming becomes even more compelling if one were to consider the increase in the complexity of managing subscriptions at the consumers as newer, finer-grained descriptions of content become available.

Hierarchical content-descriptors are intuitive, flexible and light-weight. It is quite simple to both describe content, and to sift through it. An equivalent XML-based description of the hierarchical content descriptors would be complex and heavy-weight. Hierarchical content descriptors provide an intuitive framework for finer-grained registration to stream (e.g. Streams/Sensor/Fluid/Pressure) or coarser-grained consumption (e.g. Streams/Sensor). Hierarchical streaming allows coarser grained and fine-grained consumptions to co-exist. It is up to the content dissemination network to manage the dissemination of content based on the subscriptions

1.1 Wildcards and attributes

Wildcards are placeholders specified in the subscription constraints to hierarchical streams. The wildcard operator * is used to signify placement of the wildcard. Most systems incorporate support for implicit wildcards, which correspond to the trailing portion of the hierarchical topic. Thus, the coarser-grained subscription Streams/Sensor can be formally represented as Streams/Sensor/* with the wildcard operator appearing at the end of the subscription constraint. One of the drawbacks of the implicit wildcard scheme is that a consumer may be interested in most, but not all, of the finer-grained content. To resolve this, a different type of wildcard is needed.

Wildcards can also be explicit. Such explicit wildcards can appear anywhere in the subscription constraint. By allowing more precision in the registration constraints, explicit wildcards combine the benefits of finer-grained and coarser-grained registration schemes. For example, to register an interest in fluid and atmospheric pressure readings from piezometers and barometers respectively, a consumer may register a constraint of the following form: Streams/Sensor/*/Pressure.

The scope of a wildcard operator is demarcated by the "/" operator. In the absence of a trailing "/", in the case of implicit wildcards, the wildcard operator's scope covers the entire range of the "/" separated String from that point on. Content can take on any value within the scope of the wildcard operator. A registered subscription constraint can specify multiple explicit wildcards, and will always support an implicit wildcard at the end of the constraint.

Content demarcated by "/" within the content descriptors corresponds to an attribute. The number of "/" separated attributes within a hierarchical descriptor corresponds to its depth. The depth of a hierarchical description

in turn reflects the possibilities of placing the wildcard operators, and the complexity of evaluating the corresponding subscription constraints.

Subscriptions with the leading wildcard operator * is disallowed. A stand-alone * subscription would result in all streams within the system being routed to the consumer, which would end up being deluged. From a logical perspective, if the stand-alone * subscription is allowed, all consumers regardless of whether they are interested in all streams would be deluged. Systems would also use this first attribute as the first line of defense in preventing unauthorized consumption of streams. Thus, knowledge of the first attribute would be the precursor to consuming the related streams.

1.2 Crux of this paper

In this paper we investigate strategies to organize, evaluate and enforce support for wildcards in hierarchical streaming. For hierarchical streaming, we are especially interested in three factors: performance, flux, and memory consumption. Performance is important because these streams would be produced at extremely high rates: thus, the complexity of evaluating the subscription constraints should not exceed the application's real-time thresholds. The data structures that underpin the organization scheme should be able cope with the flux inherent in streaming settings – constantly evolving interests among a large set of consumers contribute to the flux. Finally, neither the performance nor the ability to cope with flux should be at the expense of a substantial memory allocation costs associated with representing these subscription constraints.

We investigate three different algorithms. The fist one is based on graphs and is the most commonly used approach. The second one is based on using regular expressions to specify subscription constraints. Finally, we propose our algorithm based on hash tables which provides excellent performance, copes well with flux, and is optimal in its memory requirements. We have also performed extensive benchmarks, to compare and contrast the performance of these approaches.

This paper is organized as follows. Section 2 provides an overview of the NaradaBrokering content distribution network. Section 3 includes a description of the three different algorithms to organize and enforce support for wildcards in hierarchical streaming. Section 4 presents our methodology for evaluating the performance of these algorithms, and a discussion of their performance. Finally, we present our conclusions and a discussion of our proposed future work in this area.

2 NaradaBrokering

We have implemented the scheme described in this paper in the context of the NaradaBrokering [2] content distribution network, which is based on the publish/subscribe paradigm. In NaradaBrokering, this content distribution network is itself a distributed infrastructure, comprising a set of cooperating router nodes known as *broker*s. A broker performs the routing function by routing content along to other brokers within the broker network. Entities are connected to one of the brokers within the broker network, an entity uses this broker, which it is connected to, to funnel messages to the broker network and from thereon to other registered consumers of that message.

NaradaBrokering is application-independent and incorporates several services to mitigate network-induced problems as data streams traverse domains during disseminations. The system provisions these guarantees such that they are easy to harness, while delivering consistent and predictable performance that is adequate for use in real-time settings.

By specifying constraints on the content descriptors associated with individual stream fragments, consumers of a given data stream can specify, very precisely, the portions of the data stream that they are interested in consuming. The security scheme enforces the authorization and confidentiality constraints associated with the generation and consumption of secure streams while coping with several classes of denial of service attacks.

By preferentially deploying links during disseminations, the routing algorithm in NaradaBrokering ensures that underlying network is optimally utilized. This preferential routing ensures that applications receive only those portions of streams that are of interest. Since a given application is typically interested in only a fraction of all the streams present in the system, preferential routing ensures that an application is not deluged by streams that it will subsequently discard. Some of the domains that NaradaBrokering has been deployed in include earthquake science, particle physics, ecological/environmental monitoring, geosciences, GIS systems and defense applications.

3 Hierarchical Streaming

In this section we evaluate three different approaches to managing and evaluating subscription constraints in hierarchical streaming. The tree-based approach is the most commonly used approach. The regular expression based

is less commonly used. Finally, we present our algorithm, based on hashtables, which combines the benefits of the earlier approaches, without inheriting their drawbacks in memory consumption and performance overheads. For each of these algorithms, we describe the process of adding and removing subscription constraints, and also the process of computing destinations associated a stream fragment.

3.1 Tree based approach

The tree-based representation of subscription constraints on hierarchical content descriptors is the most commonly used approach. Each "/" separated subscription is first converted into a set of <*tag*, *value*> tuples. Thus, constraint of the form /Streams/Sensors/*/Pressure would be represented as the following set of comma separated <tag, value> tuples: <Tag1=Streams,Tag2=Sensors, Tag3=*, Tag4=Pressure>.

The tree representation of this subscription constraint is depicted in Figure x. One reason the Tag# is introduced is because traversal of the graph is based on the values that the edges take. By representing attribute constraints as edges in the graph, we can allow multiple edges (each corresponding to a different value for the attribute) to emerge from a node. Each edge has its own set of destinations. An edge with a destination indicates that a subscription constraint has been specified up until that point.

3.1.1 Adding and removal of subscription constraints

Subscription tuples are processed from left-to-right, and the graph traversed top-to-bottom. Nodes, and edges, are reused when possible. If an edge cannot be reused, new edges and nodes to be created from that point on, resulting in the addition of a sub-tree to the existing subscriptions tree. The last edge created as a result of processing a subscription constraint is referred to as a *destination edge*. In cases where multiple subscriptions reuse a given destination edge, their destination info appears in the destination list associated with the edge.

Each edge maintains a reference count of the number of destination edges that can be reached by traversing it. The reference count for a destination edge is the size of the destination list that it maintains. Each edge traversed during the addition (or removal) of subscriptions has its reference count increased (or decreased) by one.

During the removal of a subscription the reference counts are updated in a top-down fashion. Determination of whether edges and nodes need to be pruned from the subscriptions tree are done in a bottom-up fashion, starting at the destination edge associated with the subscription being removed. An edge is removed if its reference count is reduced to zero: this signifies that no destinations can be computed by traversing this edge. A node is removed if the last edge that originated from it is removed. Since the reference counts associated with edges closer to the root of the tree is greater than, or equal to, the reference counts associated with the child edges, if it is determined that an edge is not to be removed, no further processing of edges and nodes higher-up in the tree need to be performed.

3.1.2 Computing destinations

To compute destinations associated with a stream fragment, the content descriptors associated with stream fragment is first retrieved. These content descriptors are then used to traverse the subscription tree. At every node in the graph, the edges traversed include the edge with matching value as well as the wildcard edge. Depending on the number and placement of wildcard edges, there could be multiple traversal paths during this process.
A given traversal path may or may not end in a destination edge, but some of the edges within the path may be destination edges. The destination list for a path is the union of destination lists associated with each of the constituent destination edges. The cumulative destination list for a stream fragment is the union of the destination lists associated with each of the traversed paths within the subscription tree.

3.1.3 Complexity Analysis

While computing destinations, the worst case occurs when after the first attribute at every node the value edge as well as the wildcard edge are traversed. In the worst case, if the number of attributes is *m*, there would be $1+2 + 4 + \ldots + 2^{m-1} = \sum_{i=0}^{m-1} 2^i$ operations that would need to be performed. The complexity for computing destinations is $O(2^{m-1})$ in the worst-case. In the best case, exactly m operations would need to be performed, for a complexity of $O(m)$. Managing subscriptions typically involves the creation and deletion of nodes and links. In the worst case, for each of the N subscriptions, *m-1* nodes and *m* edges would need to be created. The space utilization in the worst case is $O(mN)$.

3.2 Regular expressions

In our second approach, we make use of regular expressions to compute destinations associated with hierarchical streaming. We first recast subscription constraints as regular expressions. To do this, we make use of the Kleene star

operator (.*) in the wildcard region demarcated by "/". In regular expression terms, the (.) corresponds to matching any single character in that position, while the (*) matches the preceding element zero or more times. In tandem, (.*) signifies that any set of characters can appear in the within its scope, which is delimited by the "/" in the subscription constraint.

3.2.1 Addition and Removal of Subscription constraints

The data structure used to store subscriptions is a Hashtable: the subscription identifier is used as the *key* and the subscription is stored as the *value* corresponding to that key. Each subscription also includes destination information. Subscription identifiers are 128-bit UUIDs (Universally Unique Identifier) to ensure that they are unique system-wide. When a subscription is added (or moved), a check is made to see if the corresponding subscription identifier is already present.

Additionally, every regular expression that is specified as a String is first compiled into a pattern, which is then used to match arbitrary character sequences against the regular expression. The Pattern engine performs traditional NFA-based (Non-Deterministic Finite-State Automata) matching.

3.2.2 Computing destinations

To compute destinations associated with a stream fragment, the content descriptors associated with stream fragment is first retrieved. Every subscription constraint (which encapsulates the regular expression query) is then matched against this identifier to determine if there is a match. In case of match, the destination within the subscription is added to the destination list associated with the fragment. As an optimization feature, a check is made to see if the subscription's destination is already present in the destination list associated with the stream fragment; if it is, the encapsulated regular expression is not evaluated.

3.2.3 Complexity Analysis

It has been shown, Ref[Katz] that the processing complexity for evaluating an NFA-based regular expression of size n is $O(n^2)$. In the worst case, where the registered subscription constraints are from different destinations, the entire set, of size N, of subscriptions would need to be evaluated. In this case, the processing complexity would be $O(n^2N)$ when assuming that n is the average size of the regular expression query. The storage overheads in this scheme correspond to storing the set of subscriptions. If there are N subscriptions, each of size n, the storage costs would be $O(nN)$.

3.3 Hashing based

In our hashing based algorithm, we try to achieve benefit from the performance gains available in the tree-based and regular expressions scheme. Specifically, we aim to have the performance of the tree-based scheme for computing destinations, but the memory utilization profile of the regular expression scheme.

Before we proceed further, we digress for a brief discussion of the location of wildcards and the number of attributes in hierarchical descriptors. As mentioned in section 1.1, subscriptions with the leading wildcard operator * is disallowed i.e. a wildcard operator cannot be specified on the first attribute of a hierarchical descriptor. This is done to prevent deluge at the subscriber, and also to ensure the authorized consumption of data streams: the first attribute would be 128-bit UUID in some cases to thwart guess and also to ensure that only authorized consumers can specify subscriptions to those streams.

To prevent the possibility of combinatorially explosive search spaces, we limit the number of attributes that can be specified within "/" separated descriptors and subscriptions. Individual attributes, however, do have any size limitations associated with them.

Normalization of subscriptions

Subscriptions are normalized at the source to remove any trailing wildcards. Thus, A/B/C* would be represented as A/B/C. Removal of implied wildcard operators simplifies the process of computing destinations.

3.3.1 Addition and removal of subscription constraints

In our algorithm, the data structure used to manage the subscriptions is the hashtable. However, in this case the subscription constraint is itself stored as the key, and the value is the destination list associated with the subscription. The first time a subscription is added to the subscriptions table, the destination list corresponding to this subscription is the destination associated with the subscription. Additional subscriptions with the same subscription constraint simply result in the addition of the corresponding destinations to the destination list associated with the subscription.

The algorithm maintains another hashtable to keep track of wildcards that have been specified. The wildcards table is indexed based on the value of the first attribute of the hierarchical descriptors. Since a wildcard is disallowed for this attribute, all subscriptions will specify this attribute. An integer array, the wildcard counts array, is then initialized with size equal to the maximum allowable number of attributes *m*.

When a subscription is added, a check is made to see if this subscription has been previously processed, and if it is stored in the subscriptions table. If the subscriptions table currently has this subscription, no further processing is done. If this is a new subscription, a check is made to determine the number and location of wildcards that have been specified within the "/" that demarcates the content descriptor attributes. Based on the value of the first attribute in the subscription constraint, an attempt is made to retrieve the wildcard counts array from the wildcards table. If an entry corresponding to the first attribute is not present in the wildcard table, a new entry is created based on the newly initialized wildcard counts array.

The wildcard counts array is incremented by one at indices corresponding to the location of wildcards. The wildcard counts array thus snapshots the locations at which a wildcard operator has been specified by subscriptions to hierarchical content descriptors.

When a subscription is removed, a check is made to determine the number and location of wildcards that have been specified for various attributes. If a wildcard is present, the wildcard counts array corresponding to the first attribute of the subscription constraint is retrieved. The wildcard counts are then decremented by one at the corresponding to the location of wildcards.

Since a wildcard cannot be specified for the first attribute, the first element in the wildcard counts array is always zero. We use this first index to keep track of the number of subscriptions that have been specified on the first attribute of the hierarchical descriptor. This is incremented the first time a subscription has been specified, irrespective of whether the constraint contains wildcard operators or not. When a subscription is removed, this count is reduced to zero indicating that this entry is ready for garbage collection.

3.3.2 Computing destinations

To compute destinations associated with a stream fragment, the content descriptors associated with stream fragment is first retrieved. Next, the wildcard counts array corresponding to the first attribute in the content descriptor retrieved. If such a wildcard counts array is not available, no subscriptions that could potentially match the content descriptor have been specified, and no further processing is performed. If, on the other hand, the wildcard counts array exists for the first attribute, processing continues.

The metadata descriptors along with indices, where the wildcards have been specified, are used to construct the set of subscriptions that would match the content descriptor. Consider the case where A/B/C/D is the content descriptor, and wildcard counts indicate that wildcards have been specified for the second and third attribute. In this case, the set of subscriptions that would be constructed would be A/B/C/D, A/*/C/D, A/B/C/* and A/*/C/* in addition to A/B.

These constructed subscriptions are then used to compute destinations associated with the stream fragment. For every subscription, a simple lookup of the subscriptions table yields the corresponding destination list. The destination list for the stream fragment is the union of the destination lists associated with each of the constructed subscriptions.

3.3.3 Complexity Analysis

Two hashtables: one for wildcard counts and another for maintaining destination lists. For a set of N subscriptions the space utilization for each of these tables is *O(*N*)* in the worst case where each subscription constraint is unique.

Access times Hashtables is O(1). In the best case, only one such access would be needed to retrieve the destinations list. In the worst case, for hierarchical descriptors with a maximum of the m attributes and wild card operators for every attribute except the first one, 2^{m-1} O(1) accesses need to be made.

Expansion of the hashtable is quite expensive. To keep these costs under control, the initial capacity of the Hashtable is set to preclude the need for unnecessary expansions. The hashtables are initialized with a load factor of 0.75. The expected number of probes needed when a collision occurs is 1/(1-loadFactor); thus, 4 probes are needed on average during collisions. The initial capacity controls a tradeoff between wasted space and the need for `rehash` operations, which are time-consuming.

The memory consumption is approximately O(*N*). In the worst case, all the *N* registered subscription constraints would be unique. If each registered subscription constraint is of size *n*, the memory consumption would be O(*nN*).

4 Performance Evaluation

In this section we describe or experimental methodology and a discussion of results obtained from our benchmarks.

4.1 Experimental Methodology

We first start-off by presenting results outlining the communication latencies in a simplified setting involving one producer and consumer. The communication latencies will be reported for stream fragments with different payload sizes, each of which has a one-attribute content descriptor. Since these communication latencies include the time spent in computing destinations, we expect that this would give the reader a good feel what would constitute as an acceptable overhead in computing destinations. In general, the overheads introduced by a good algorithm for computing destinations would be a fraction of the end-to-end communication costs in cluster settings. Readers who are interested in NaradaBrokering benchmarks in settings involving broker networks are referred to [][][].

To benchmark the three algorithms for hierarchical streaming we profile several aspects related to its performance, ability to cope with flux and memory utilization. To measure the performance of the algorithms, we vary the number of attributes in the content descriptors and the corresponding subscription constraints. Additionally, we also vary the number of subscriptions that are managed by each algorithm. Under conditions of varying number of attributes and the number of subscriptions, we compute the costs involved in computing destinations for a given stream fragment. These calculation costs reveal the suitability of each algorithm for real-time streaming.

To determine the ability of the algorithms to cope with flux, we also compute the costs involved in adding and removing subscriptions for different subscription sizes. We also profiled the tree-based approach for its memory utilization: specifically, we track the number of nodes that are created for different number of attributes as the number of subscriptions varies.

4.2 Experimental Results

Our first set of benchmarks relate to measuring stream communication latencies in a cluster settings. We benchmarked the simplest case, where the content distribution network comprises a broker, a producer and a consumer. There is just one subscription being maintained, and is specified on a content descriptor with exactly one attribute. This setting will reveal the lowest possible overheads for streaming in LAN settings.

The two cluster machines (4 CPU, 2.4GHz, 2GB RAM)) involved in the benchmark were hosted on 100 Mbps LAN. The producer and consumer were hosted on the same machine to obviate the need to account for clock drifts while measuring latencies for streams issued by the producer, and routed by the broker (hosted on the second machine) to the consumer. All processes executed within version 1.6 of Sun's JVM.

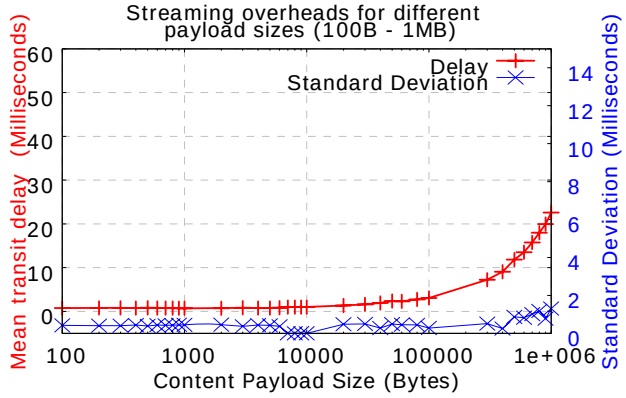

Figure 1: Streaming overheads in cluster settings

The results, depicted in Figure 1, report the mean communication delays for different payload sizes encapsulated within the stream fragments. The reported delay is the average of 50 samples for a given payload size, with the standard deviation for these samples also being reported. For stream fragment payload sizes, the delays are around a millisecond for payloads up to a 10 KB, and increasing to 20 milliseconds for 1 MB payload size. It must be noted that in WAN settings the communication latencies are in the order of tens of milliseconds and can go up to a 130 millisecond per hop.

The remainder of the benchmarks, pertain to the three algorithms presented in this paper, and were performed on a standalone machine (4 CPU, 2.4GHz, 2GB RAM). We used a high-resolution timer to report several of our measurements in microseconds.

To measure the performance of the algorithms, we vary the number of attributes in the content descriptors and also the corresponding subscriptions. Additionally, we vary the number of subscriptions being managed by each algorithm from 10^4 to 10^5 subscriptions. The subscriptions are generated randomly, with every attribute being randomly assigned one of 50 possible values. For each subscription, except for the first attribute, wildcards will be specified on at least one of the other attributes.

Figures Figure **2**, Figure **3** and Figure **4** depict the overheads for computing destinations in tree-based, hashing and regular expressions scheme. In general, the matching overheads increase as the number of subscriptions and the number of attributes within the subscriptions increase. It must be noted that given the large number of subscriptions, and also that these subscriptions are generated randomly, for a given hierarchical descriptor (based on the first

attribute), a wildcard operator eventually appears for every other attribute. This in turn causes the hashing-based scheme – Figure Figure **3** – to approach its worst-case performance wherein the number of sweeps of the Hashtable become proportional to the number of attributes. In the regular expressions case, Figure Figure **4**, do not depart significantly from their high base costs.

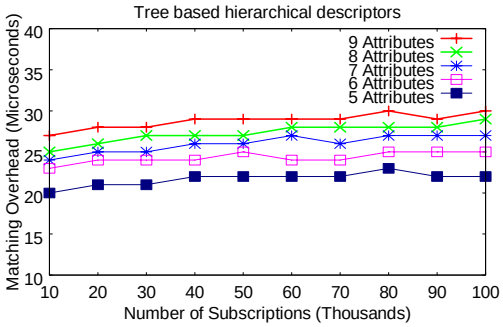

Figure 2: Overheads for tree-based scheme

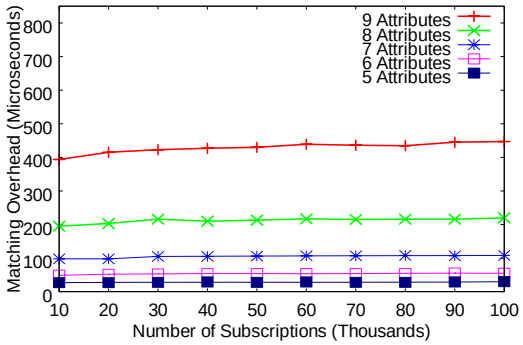

Figure 3: Overheads for the Hashing scheme

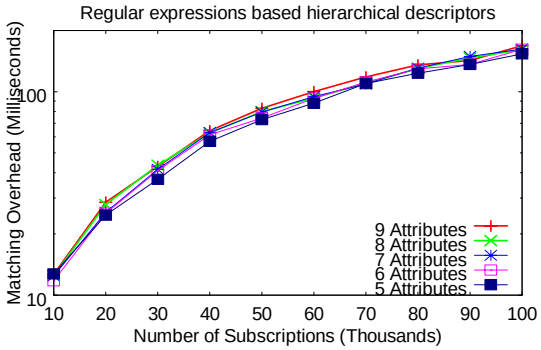

Figure 4: Overheads for Regular expressions

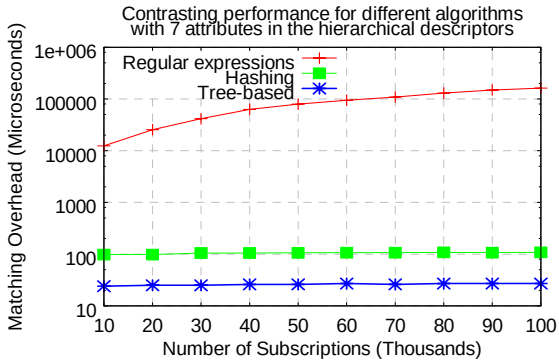

Figure 5: Overhead comparisons for different algorithms with 7 attributes

Figure 5 contrasts the matching overheads for the three algorithms for varying number of subscriptions, each of which have 7 attributes. It is clear that the matching overheads are the best in the case of the tree-based and hashing-based schemes, while the overheads introduced by the regular expressions scheme are several orders of magnitude higher than the other two.

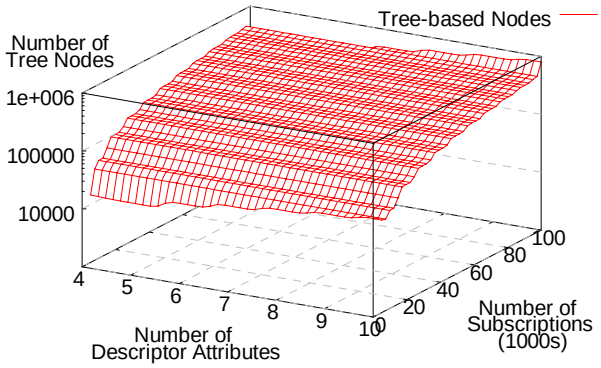

Figure 6: Node allocation costs in tree-based scheme

Perhaps the biggest drawback of the tree-based scheme is the memory requirements associated with maintaining the subscriptions. Figure 6 depicts the memory allocation costs associated with the tree-based scheme. As the number of attributes and subscriptions increase, the number of nodes and edges needed to represent the set of managed subscriptions also increase substantially. Case in point is the fact that for managing 100000 subscriptions, each with 10 attributes, results in the creation of 798188 nodes and 898187 edges: approximately 2 million objects! During the benchmark process the heap size allocated for the JVM had to be set to more than 1 GB to benchmark the tree-based scheme.

We also performed benchmarks to determine the ability of the algorithms to cope with flux, wherein subscriptions are being added and removed at high rates. Figure 7 and Figure 8 depict the cost associated with adding and removing one subscription for each of the algorithms. The regular expressions scheme delivers the best performance, with the hashing-based performance quite close to this. The additional overhead in the hashing scheme

is introduced by the need to maintain the wildcard counts array. The higher costs in the tree-based scheme pertain to the creation or removal of nodes and edges.

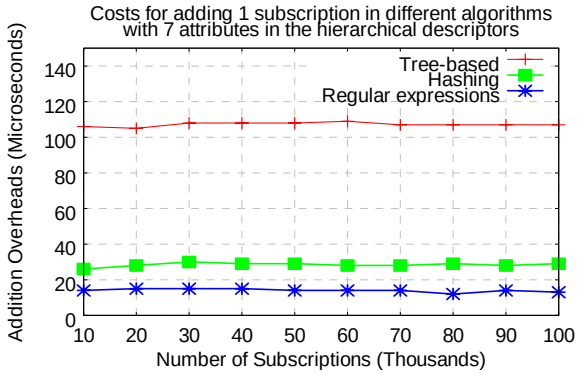

Figure 7: Costs for adding a subscription

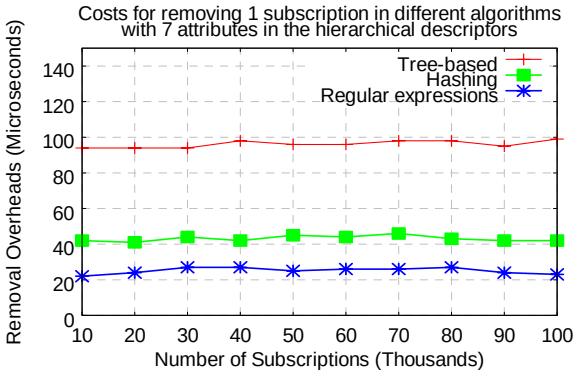

Figure 8: Costs for removing a subscription