Enabling Hierarchical Dissemination of Streams in Content Distribution Networks

Shrideep Pallickara
Department of Computer Science,
Colorado State University
shrideep@cs.colostate.edu

Geoffrey Fox
Department of Computer Science,
Indiana University
gcf@indiana.edu

Abstract— **In streaming systems the content distribution network routes streams based on interests registered by the consuming entities. In hierarchical streaming, the dissemination is also predicated on the resolution of hierarchical dependencies between various streams. Entities specify explicit wildcards, in addition to the implicit ones in place, to further control the types of streams within a given hierarchy that should be routed to them. This paper presents an analysis and performance evaluation of three different algorithms for hierarchical streaming. In our evaluation of these algorithms we are especially interested in three factors: performance, ability to cope with flux, and memory consumption. Comprehensive benchmarks for these algorithms, in this paper, will enable system designers to harness the best algorithm that satisfies their hierarchical streaming requirements.**

Keywords- streaming systems, hierarchical streaming, content distribution networks, pub/sub, middleware

I. INTRODUCTION

Streaming pertains to the routing of data streams from the sources to entities that are interested in them. 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-to-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* that are constraints specified on the values that the content descriptors might take. Subscription complexity is directly proportional to the richness of the content description. Dissemination of streams is based on these subscriptions and the stream's content descriptors.

The simplest content descriptor is a character String, for e.g. Streams/Sensor. This simplicity 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 increased complexity of managing subscriptions at the consumers as newer, finer-grained descriptions of content become available.

Hierarchical content-descriptors are intuitive, flexible and lightweight. It is quite simple to describe and sift through content. Hierarchical streaming allows coarser-grained (e.g. Streams/Sensor) and fine-grained (e.g. Streams/Sensor/Fluid/Pressure) consumption patterns to co-exist. An equivalent XML-based description of the hierarchical content descriptors would be complex and heavyweight.

A. Wildcards and attributes

Wildcards, denoted by *****, are placeholders specified in the subscription constraints to hierarchical streams. Most systems incorporate support for implicit wildcards, whose scope is over the *trailing* portion of the hierarchical descriptor. Thus, the coarser-grained subscription Streams/Sensor is equivalent to Streams/Sensor/* with the wildcard 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 content that would then be routed to it.

Wildcards can also be explicit. Such explicit wildcards can appear anywhere in the subscription constraint. By allowing more precision in the registration of 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 "/" in the hierarchical descriptors; for implicit wildcards, the scope begins at the end of the subscription constraint. Content can take on any value within the scope of the wildcard. A registered subscription constraint can specify multiple explicit wildcards, and will always have an implicit wildcard at the end. There could be situations where an entity is not interested in hierarchical streaming; for example, embedded devices often need to be explicit about the data that they receive. This can be achieved by (1) encoding type information into topics that allow them to be demarcated as regular or hierarchical , and (2) by using another wildcard (#) as an indicator that hierarchical streaming should be disabled for that subscription. The focus of this paper is hierarchical streaming.

Content demarcated by "/" within the content descriptors corresponds to an *attribute*. The number of "/" separated attributes within a hierarchical descriptor is its *depth*. The depth of a hierarchical description in turn reflects the number of possibilities of placing wildcard operators, and the complexity of evaluating specified subscription constraints.

A subscription with a wildcard on the first attribute is disallowed. A stand-alone * subscription would result in all streams within the system being routed to the consumer, which would then end up being deluged. Disallowing use of the * wildcard in the first attribute not only restricts the search space during matching operations but also accounts for privacy in cases where subscriptions that rely on UUIDs (with a 2^{128} ID space) for encoding their first attribute are assured that their streams are not consumed by such an all-encompassing * subscription.

B. Crux of this paper

In this paper we focus on managing subscription constraints and computing destinations based on hierarchical content descriptors encapsulated in individual stream fragments. Once the destinations have been computed it is the responsibility of the content dissemination network to efficiently disseminate these streams by calculating routes to reach these destinations. Our previous work, Pallickara et al [1], describes a routing algorithm, which ensures that the computed routes are efficient and avoid intermediate nodes that have failed or have been failure-suspected.

Specifically, we investigate strategies to organize, evaluate and enforce support for wildcards in hierarchical streaming. For hierarchical streaming, we are especially interested in three factors: computational performance, flux, and memory consumption. Since streams would be produced at high rates, the complexity of evaluating subscription constraints should not exceed an application's real-time threshold; for example, an application may tolerate stream dissemination delays that are in the order of a few milliseconds, if the processing overhead related to computing destinations is high the overall dissemination delays might be unacceptable to the application. Processing overheads also result in bottlenecks resulting in queue build-ups that might strain that buffers at the intermediate nodes. Data structures that underpin the organization scheme should be able cope with the inherent *flux* caused by constantly evolving interests among a large set of consumers that result in high registration and deregistration of subscriptions. Finally, neither the performance nor the ability to cope with flux should be at the expense of substantial memory allocation costs associated with representing these subscription constraints.

C. Paper Contribution

This paper presents an analysis and performance evaluation of three different algorithms for hierarchical streaming. Algorithms for computing destinations for hierarchical streaming tend to be either tree-based, which are computationally optimal but memory intensive, or are regular-expressions based, which make optimal use of memory but with poor response times. The asymptotic complexity of a hashing-based algorithm that we have designed matches that of the tree-based case for computational efficiency, and that of the regular expressions case for memory utilization. We have performed extensive benchmarks, to compare and contrast these algorithms and they confirm the suitability of our algorithm and its ability to cope with flux.

The primary contribution of this paper is that it will enable system designers to make informed decisions about the algorithm that best satisfies their hierarchical streaming requirements.

D. Applicability of Hierarchical Streaming

Hierarchical streaming is particularly suitable for managing disseminations in several domains; here, we focus on three such domains: workflows, map-reduce enabled applications, and networked observational environments.

In workflows, the outputs of consecutive stages of the pipeline can successively add attributes to the content descriptors signifying the outputs of different stages. A given computational unit could be part of different stages within a pipeline or multiple workflows. A stage may also find it appropriate to rewrite the content descriptors after a processing step.

Map-reduce is a framework utilized in cloud computing wherein the processing of large datasets is split into smaller components (*map*s) that process smaller portions of the datasets, the results of which are then combined (*reduce*) to reconstitute the final result. These map-reduce operations can be sequential or iterative. Hierarchical streaming can be used to

not only collate results produced by individual map functions, but also to identify, process and fuse outputs produced by different iterations of a given map-reduce computation.

In networked observational environments, data produced by sensing equipment needs to be routed to different computational units depending on the hardware, metric, and precision of the data. Additionally, these observational systems need to incorporate support for the addition and removal of sensing equipment without having to update the processing units at disparate locations. Hierarchical streaming can enable selective routing and also manage the flux in the devices being used in observational settings.

Paper Organization: Section II provides an overview of the NaradaBrokering content distribution network. Section III includes a description of the three different algorithms to organize and enforce support for wildcards in hierarchical streaming. Section IV presents our performance evaluation. In Section V we describe related work in this area. Finally, we present our conclusions and a discussion of our proposed future work in this area.

II. NARADABROKERING

We have implemented the algorithms described in this paper in the context of the NaradaBrokering [1,2] content distribution network that we have developed. The NaradaBrokering content distribution network comprises a set of cooperating router nodes known as *broker*s. Entities connect to one of the brokers within the broker network, and use their hosting broker to funnel streams into the broker network and from there-on to other registered consumers of those streams.

NaradaBrokering incorporates several services to mitigate network-induced problems as streams traverse domains during disseminations. The system provisions reliability and ordering guarantees, while delivering consistent and predictable performance that is adequate for use in real-time settings. In NaradaBrokering entities specify constraints on the qualities of service (QoS) related to the delivery of streams. The QoS pertain to reliable delivery, playbacks, order, duplicate elimination, global timing services, security, and size of the published stream fragments and their encapsulated payloads.

By specifying constraints on the content descriptors associated with individual stream fragments, consumers of a given data stream can specify, very precisely, portions of the data stream that they are interested in consuming. The system enforces [2] authorization and confidentiality constraints associated with the generation and consumption of secure streams while coping with certain classes of denial of service attacks.

By preferentially deploying links during disseminations, the routing algorithm in NaradaBrokering ensures that the underlying network is efficiently 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 the streams present in the system, preferential routing ensures that an application is not deluged by streams that it will subsequently discard. The routing of streams within the broker network is also preferential: we do not resort to flooding the broker network. Brokers only route content to brokers that contain valid subscriptions or those that are on the path to such targeted brokers. Furthermore, every broker, either targeted or en route to one, computes the shortest path to reach target destinations while eschewing links and brokers that have failed or have been failure- suspected.

Some of the domains that NaradaBrokering has been deployed in include earthquake science, particle physics, environmental monitoring, geosciences, GIS systems, and defense applications.

III. HIERARCHICAL STREAMING

In this section we describe three different approaches to managing and evaluating subscription constraints in hierarchical streaming. The tree-based approach is the more commonly used approach, while the regular expression based approach is less commonly used. We also present our algorithm, based on hashtables. For each algorithm, we describe the addition/removal of subscription constraints, and calculation of destinations for stream fragments.

A. Tree based approach

In the tree-based representation of subscription constraints each "/" separated subscription is first converted into a set of comma separated<*tag*=*value*> tuples. Thus, a constraint of the form `/Streams/Sensors/*/Pressure` would be represented as the following: `<Tag1=Streams, Tag2=Sensors, Tag3=*, Tag4=Pressure>`. The tree representation of this subscription constraint, within an existing subscription tree, is depicted in Figure 1.

The Tag# is introduced 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 of 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 until that point.

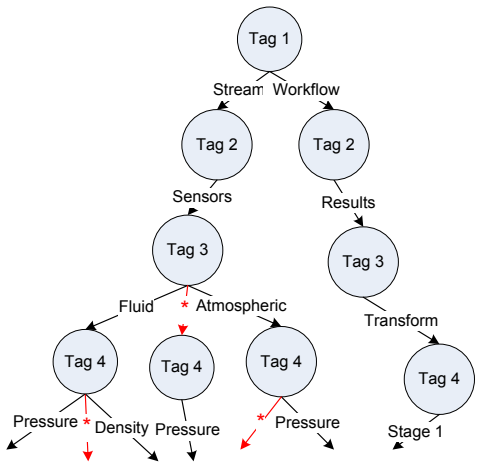

Figure 1: An example subscription tree

1) Adding and removal of subscription constraints

When processing subscription constraints the tree is traversed from top-to-bottom. Nodes and edges are reused when possible. If an edge cannot be reused, new edges and nodes will 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*. When multiple subscriptions reuse a given destination edge, the corresponding destination info appears in the destination list associated with that 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.

Determining whether edges and nodes need to be pruned from the subscription tree is 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 reduces 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. Reference counts associated with edges closer to the root of the tree are greater than, or equal to, the reference counts associated with the child edges. So, if it is determined that an edge is not to be removed, pruning of edges and nodes higher-up in the tree is not needed.

2) Computing destinations

To compute destinations associated with a stream fragment, the corresponding content descriptors are first retrieved. This is then used to traverse the subscription tree. At every node at most 2 edges may be traversed: the edge with matching value and, if present, the wildcard edge. Depending on the number and location of wildcard edges, there could be multiple traversal paths during this process.

A given traversal path may include zero or more 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.

3) Complexity Analysis

While computing destinations, the worst case occurs when after the first attribute at every subsequent node 2 edges – the value edge and 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, each of cost $O(1)$, need to be performed. The complexity for computing destinations is $O(1)$ where the constant is 2^{m-1} in the worst-case. In the best case, exactly m operations would need to be performed, for a complexity of $O(1)$ where the constant is 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 $O(N)$ where the constant is m.

B. 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. The (**.***) in tandem signifies that any set of characters can appear within the wildcard's scope.

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 corresponding *value*. Subscriptions include destination information. Subscription identifiers are 128-bit UUIDs (Universally Unique Identifier) to ensure system-wide uniqueness, and are used during the addition and removal of subscriptions to see if a subscription was previously registered.

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 (Non-Deterministic Finite-State Automata) matching.

2) Computing destinations

To compute destinations associated with a stream fragment, the corresponding content descriptors are first retrieved. Every subscription constraint (encapsulating the regular expression query) is then evaluated against this identifier to determine if there is a match. If there is a 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) Complexity Analysis

It has been shown, Yu et al [3], 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 all 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. In general, if there are M distinct destinations the processing complexity is $O(n^2M)$. The storage overheads in this scheme correspond to storing the set of subscriptions. If there are N subscriptions, the storage complexity is $O(N)$ with a fixed small constant that is independent of the number of attributes.

C. Hashing based

In the hashing based algorithm, we aim to have the performance of the tree-based scheme for computing destinations, but the memory utilization profile of the regular expression scheme.

1) Addition and removal of subscription constraints

The data structure used to manage the subscriptions is the hashtable. The subscription constraint is itself stored as the *key*, and the *value* is 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 the first attribute, all subscriptions will specify this.

```
MANAGESUBSCRIPTIONADDITION(A, consumerDest)
  INITIALIZEWILDCARDCOUNTSARRAY(A1)
  wcounts = GETWILDCARDCOUNTSARRAY(A1)
  for i ← 2 to SIZE(A)
    if Ai = *
      then wcounts[i] ← wcounts[i] + 1
  ADDSUBSCRIPTION(A, consumerDest)
  wcounts[1] ← wcounts[1] + 1

ADDSUBSCRIPTION(A, consumerDest)
  if subscription A in dictionary
    then dest ← get destinations from subscription dictionary
         dest ← dest U consumerDest
    else put (A, consumerDest) into subscription dictionary
```

```
INITIALIZEWILDCARDCOUNTSARRAY(attribute)
  if attribute in wildcard dictionary
   then return
   else  wcounts = ALLOCATE(maxAttributeDepth)
        put (attribute, wcounts) into wildcard dictionary

GETWILDCARDCOUNTSARRAY(attribute)
  return retrieved counts from wildcard dictionary
```

Figure 2: Algorithm for managing addition of subscriptions

When a new subscription needs to be processed (depicted in Figure 2), the subscription constraint attributes are processed before the subscription can be added to the subscriptions-table. 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 wildcards-table, a new entry is initialized with the maximum allowable number of attributes *m*. Next, we determine the number and location of wildcards that have been specified within the "/" that demarcate the content descriptor attributes. The wildcard-counts array is incremented by one at the indices corresponding to the location of wildcards. The wildcard-counts, for the first attribute of a hierarchical descriptor, thus snapshots the locations at which wildcards have been specified by the set of related (similar first attribute) subscriptions.

The first time a subscription is added to the subscriptions-table, the destination list for this subscription is initialized to the destination associated with the subscription. Additional subscriptions with the same subscription constraint result in the addition of the corresponding destinations to that subscription's destination list.

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 indices corresponding to the location of the 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, with a matching first attribute, has been specified irrespective of whether the constraint contains wildcard operators or not. Removal of the subscription will result in a corresponding reduction in the count. When the subscription-count corresponding to the first attribute is reduced to zero, the space allocated for the wildcard-counts array will be reclaimed.

2) Computing destinations

To compute destinations (depicted in Figure 3) associated with a stream fragment, the corresponding content descriptors are 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 the wildcard counts array exists for the first attribute, processing continues.

```
COMPUTEDESTINATIONS(A)
  dest ←-NIL, level ← 1
  wcounts ← GETWILDCARDCOUNTSARRAY(A1);
  if (wcounts = NIL)
   then return dest

  dest ← GETDESTINATIONFOR(A1);
  dest ← dest U FINERRECURSION (dest, A, wcounts, A1, level)

FINERRECURSION (dest, A, wcounts, coarserSub, level)
  if (level > SIZE(A))
   then return dest

  level ← level + 1
  finerSub ← coarserSub + "/" + Alevel
```

```
dest ← GETDESTINATIONFOR (finerSub)
dest ← dest U FINERRECURSION (dest, A, wcounts, finerSub, level)

if (wcounts[level] > 0)
 then finerWCSub ← coarserSub + "/*"
      dest ←GETDESTINATIONFOR (finerWCSub)
      dest ←dest U FINERRECURSION(dest, A, wcounts, finerWCSub,level)

return dest

GETDESTINATIONFOR (subscription)
  Perform dictionary operation to retrieve destination
```

Figure 3: Algorithm for computing destinations

The content 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 are: A/B/C/D, A/*/C/D, A/B/C/* and A/*/C/* in addition to A and 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) Complexity Analysis

The complexity of supporting dictionary operations for a hashtable on the average is *O(1)*. Thus, the lookup, addition and retrieval times for a hashtable is O(1). When computing destinations, in the best case, only one such access would be needed to retrieve the destinations list for the subscription constraint. 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} accesses (each with a cost of *O(1)*) would need to be made. The *O(1)* costs in our hashtable scheme would be slightly higher than the corresponding O(1) costs in the tree-based scheme: our benchmarks confirm this. The memory consumption is O(*N*) in the worst case, when all the *N* subscription constraints are unique. The constant for the space-complexity would depend on the implementation strategy: the Google Sparse Hash, for example is extremely memory-efficient with only a 2 bit overhead per entry. In our implementation and benchmarks we used the hashtable that is available as part of the Java libraries.

IV. PERFORMANCE EVALUATION

We first start-off by presenting results outlining the communication latencies in a simplified setting involving one producer and consumer. The communication latencies are reported for stream fragments with different payload sizes, each of which has a one-attribute content descriptor. The reported communication latencies include the time spent in computing destinations. NaradaBrokering benchmarks in settings involving broker networks can be found in [1,2].

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 *both* the number of attributes in the content descriptors and the number of subscription constraints that are managed by each algorithm. We then report the costs involved in computing destinations for a given stream fragment.

To determine the ability of the algorithms to cope with flux, we compute the costs involved in adding and removing subscriptions when the size of the managed subscriptions vary.

A. Streaming in Cluster Settings

Our first set of benchmarks relate to measuring stream communication latencies in cluster settings. We benchmarked the simplest case involving one producer, one consumer, and a content distribution network that comprises one broker. There is just one subscription being maintained, and it is specified on a content descriptor with exactly one attribute. This setting reveals the lowest possible latencies for streaming in LAN settings. For real-time streaming, in multimedia settings, the acceptable latencies are typically about 10-30 milliseconds in LAN settings, and around 100-200 milliseconds in WAN 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.

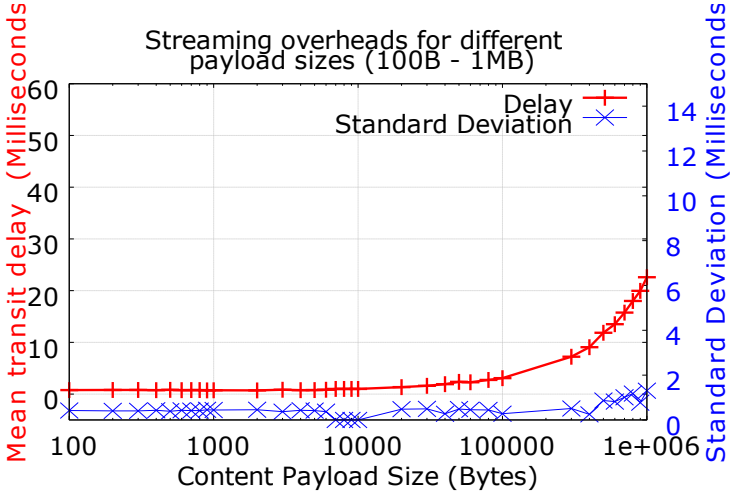

Figure 4: Streaming overheads in cluster settings

The results, depicted in Figure 4, report the mean communication delays for different payload sizes encapsulated within the stream fragments. The reported delay is the average of 50 experimental runs for a given payload size; 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.

B. Performance of the algorithms

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) with processes executing within version 1.6 of Sun's JVM.

1) Computational Performance

To measure the computational performance of the algorithms, we vary the number of attributes in the content descriptors and also the number of managed subscriptions in 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 one of the other attributes.

The choice of the number of attributes and the sizes of the subscription tables are based on our experience with actual deployments of NaradaBrokering in environmental and internet conferencing based systems. (http://www.naradabrokering.org/deployments/index.html)

Each point in our graphs (figures 5, 7 and 8) corresponds to the average of a 100 experimental runs on a dedicated machine on which no other user jobs were executing. The standard deviations involved in these measurements were low: for computing destinations, in the tree-based case it was around 1 microsecond while in the hashing scheme it was around 4-10 microseconds.

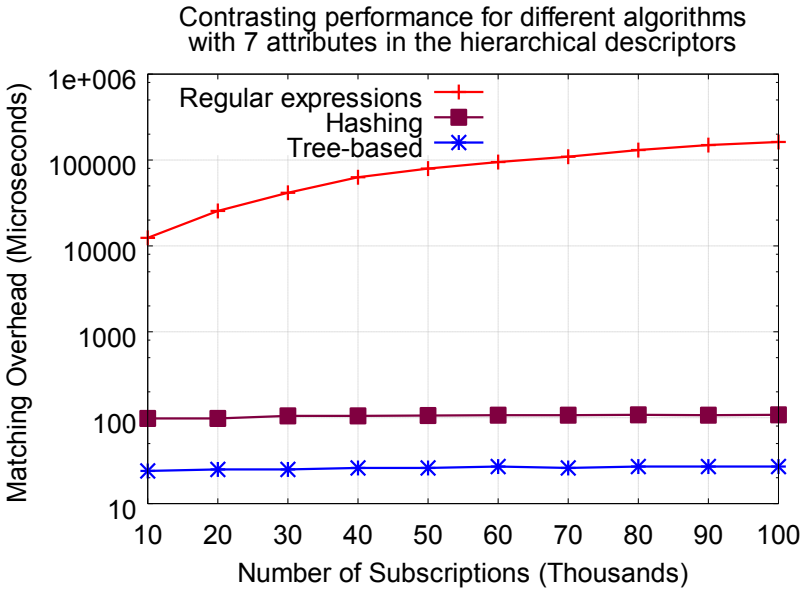

Figure 5: Cumulative overhead comparisons

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 scheme, with slightly higher overheads for the hashing-based scheme. The overheads introduced by the regular expressions scheme are several orders of magnitude higher than that of the other two.

2) Space Utilizations

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 in the tree-based case, managing 100000 subscriptions, each with 10 attributes, results in the creation of 798188 nodes and 898187 edges: approximately 2 million objects. During the benchmarks, the heap size allocated for the JVM had to be set to more than 1 GB for the tree-based scheme. In contrast, for the hashing and regular expression based schemes the total number of entries was equal to the number of unique subscriptions that were specified; a number that was two orders of magnitude lower than the number of tree nodes in the tree-based scheme.

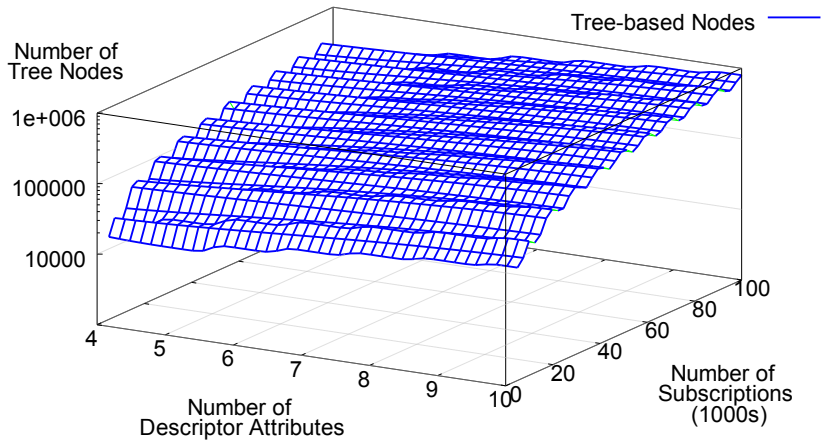

Figure 6: Node allocation costs in the tree-based algorithm

The space utilizations in both the Regular expressions and hashing based approaches corresponds to the number of distinct subscription constraints and is thus optimal. In our benchmarks for the regular expression and hashing based approaches the heap size for the JVM was set to 64 MB.

3) Coping with flux

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.

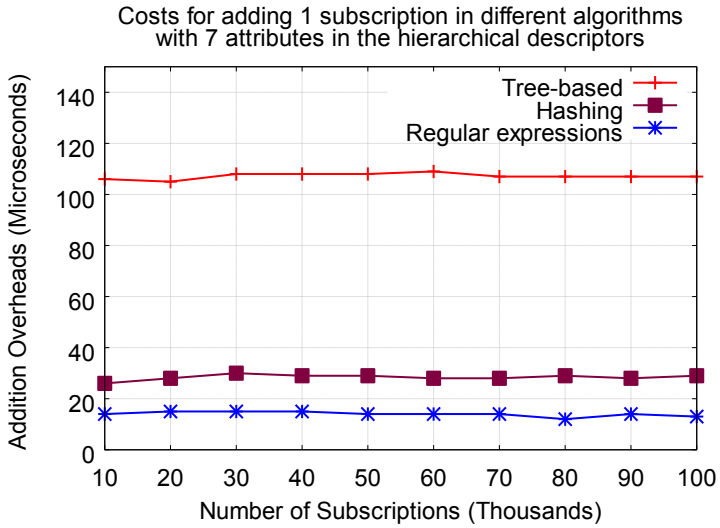

Figure 7: Costs for adding a subscription

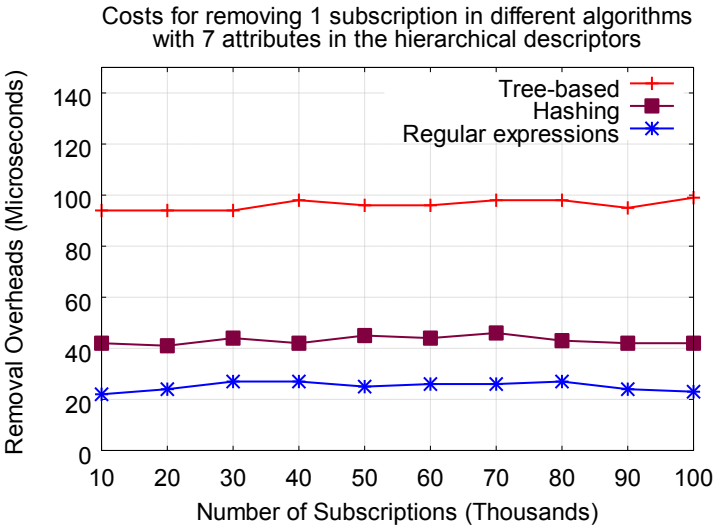

Figure 8: Costs for removing a subscription

V. Related Work

Support for tree-based <tag,value> tuples with equality checks and wildcards in the values was first used in the Gryphon [6] system. Gryphon's matching scheme provides a time-complexity that is sub-linear in the number of subscriptions. However, even though their complexity of space consumption is linear in the number of subscriptions, the constant is high enough that the costs become prohibitive as the number of attributes increase. An optimization to their matching algorithm based on successor nodes, reduces the matching time even further by 20%, but at the expense of increased space complexity. Their suggested space optimization involves collapsing chains of *-edges will not have a significant effect: in our benchmarks, where we randomly generated constraints, there were no subscription constraints that lead to such * chains and the space costs were still high (Figure 6).

The WS-Topics [5] specification incorporates support for organizing topics and also for maintaining aliases associated with these topics. While wildcards are not explicitly supported, subscribers can navigate the topic hierarchy to determine the topics to subscribe to. WS-Topics is part of the Web Service Resource Framework (WSRF) suite of specifications that are used to build Grid systems. WSRF is a realignment of the dominant Open Grid Service Infrastructure [6] with the Web Services community. Research has also been done on automated negotiations [7] of QoS between publishers and subscribers in Grid notifications settings; the choice of the hierarchical streaming algorithm could conceivably be part of such negotiations.

Segall et. al. [8] outline a strategy to convert each subscription in Elvin into a deterministic finite state automaton. This conversion, and the matching solutions, can lead to an combinatorial explosion in the number of states for a small number of subscriptions. Systems such as SonicMQ [9] and TIBCO [10] incorporate support for hierarchical "/"-separated topic spaces with support for wildcard operators. However, there is no published literature that describes their algorithms, concomitant complexity, and targeted trade-off space.

The Java Message Service (JMS) [11] specification from Sun defines a set of Java interfaces that enables the development of publish/subscribe applications. Individual messages have properties associated with them. Constraints based on SQL queries can specified on the values that these properties take. SQL-queries and regular expressions solve different types of problems; regular expressions are useful for representing patterns which are difficult to represent in SQL, which in turn can be much more expressive for expressing selection criteria.

The Event Service [12] approach adopted by the OMG is one of establishing channels and subsequently registering suppliers and consumers to the event channels. The approach could entail clients (consumers) to be aware of a large number of event channels.

VI. Conclusions

Hierarchical descriptors provide a flexible, lightweight scheme for content description and also for the specification of constraints on these content descriptors. In this paper we presented algorithms that could be utilized for enabling hierarchical streaming.

Regular expressions provide a rich language for the specification of constraints through various operators that enable specification of patterns, partial matches, placeholders, and case independence among others. However, the computational costs introduced by the regular expressions scheme can be prohibitive as the number of subscription constraints increase.

The tree-based approach provides excellent performance, but the memory costs associated with maintaining the nodes and edges associated with individual subscription constraints increase substantially as the number of the attributes and subscriptions increase. In our benchmarks, for 10^5 subscriptions each with 10 attributes, about 2 million elements (edges and nodes combined) were created.

The hashing-based scheme provides performance approaching that of the tree-based scheme while at the same time providing excellent memory utilization performance.

The regular-expressions based scheme provides the best performance while coping with flux in the set of managed subscriptions. The performance of the hashing-based scheme is closer to that of the regular-expressions based scheme, while the tree-based approach was the slowest of the three.

We are hopeful that this work will enable system designers to make informed decisions about the algorithm that best satisfies their hierarchical streaming requirements.

As part of our future work, we will investigate the use of hierarchical streaming in map-reduce style computations both in the single-phase and iterative modes. This will be the subject of our future papers in this area.

REFERENCES

1. S Pallickara et al. A Framework for Secure End-to-End Delivery of Messages in Publish/Subscribe Systems. Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID 2006). Barcelona, Spain.
2. S Pallickara and G Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of the ACM/IFIP/USENIX International Middleware Conference Middleware-2003. pp 41-61.
3. F. Yu, Z. Chen, Y. Diao, T. Lakshman and R. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems.
4. Marcos Aguilera et al. Matching events in a content-based subscription system. In Proceedings of the 18th ACM Symposium on Principles of Distributed Computing Systems.1999.
5. Web Services Topics (WS-Topics). IBM, Globus, Akamai et al.ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-Topics.pdf
6. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration." Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.
7. Richard Lawley, Michael Luck, Keith Decker, Terry R. Payne, Luc Moreau: Automated Negotiation Between Publishers And Consumers Of Grid Notifications. Parallel Processing Letters 13(4): 537-548 (2003).
8. Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with Elvin4. In Proceedings AUUG2K, Canberra, Australia, June 2000.
9. SonicMQ: Enterprise Messaging System: www.sonicsoftware.com/
10. P Maheshwari, M Pang: Benchmarking message-oriented middleware: TIB/RV versus SonicMQ. Concurrency - Practice and Experience 17(12): 1507-1526 (2005)
11. M. Happner, R Burridge and R Sharma. Sun Microsystems. Java Message Service Specification. 2000.
12. The Object Management Group (OMG). OMG's CORBA Event Service. Available from http://www.omg.org/