Enabling Hierarchical Dissemination of Streams in Content Distribution Networks

Shrideep Pallickara (spallick@indiana.edu)
Community Grids Lab, Indiana University

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. We investigate three approaches to organize, evaluate and enforce support for wildcards in 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, which satisfies their hierarchical streaming requirements.

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

1 Introduction

Streaming pertains to the routing of data streams from the sources to 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-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 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 lightweight. It is quite simple to describe content, and also to sift through it. An equivalent XML-based description of the hierarchical content descriptors would be complex and heavyweight. Hierarchical content descriptors provide an intuitive framework for consumption patterns that could be finer-grained (e.g. `Streams/Sensor/Fluid/Pressure`) or coarser-grained (e.g. `Streams/Sensor`). Hierarchical streaming allows coarser-grained and fine-grained consumption patterns to co-exist.

1.1 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. 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 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.

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. Systems may wish to reserve the first attribute to prevent unauthorized consumption of streams. Here, knowledge of the first attribute would be the precursor to consuming the related streams. Of course, additional cryptographic operations would need to be performed to ensure that the disseminations are indeed authorized.

1.2 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, Ref [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. 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. 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.

We investigate three different algorithms. The first one, and the most commonly used, is tree-based. The second one is based on using regular expressions for subscriptions. Finally, we propose our algorithm based on hashtables.

1.3 Paper Contribution

The primary contribution of this paper is an algorithm, for computing destinations in hierarchical streaming, whose memory consumption and computational overhead is very efficient. 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 our algorithm 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.

1.4 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. 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 equipments need 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 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 performance evaluation. In Section 5 we describe related work in this area. 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 [1,2] content distribution network. The NaradaBrokering content distribution network comprises a set of cooperating router nodes known as *broker*s. Entities, connected to one of the brokers within the broker network, use their hosting broker to funnel streams into the broker network and from thereon to other registered consumers of those streams.

NaradaBrokering is application-independent and incorporates several services to mitigate network-induced problems as streams traverse domains during disseminations. The system provisions easy to use guarantees, 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 [2] 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 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 describe three different approaches to managing and evaluating subscription constraints in hierarchical streaming. The tree-based approach is the most 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 and removal of subscription constraints, and computing destinations for stream fragments.

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 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 up until that point.

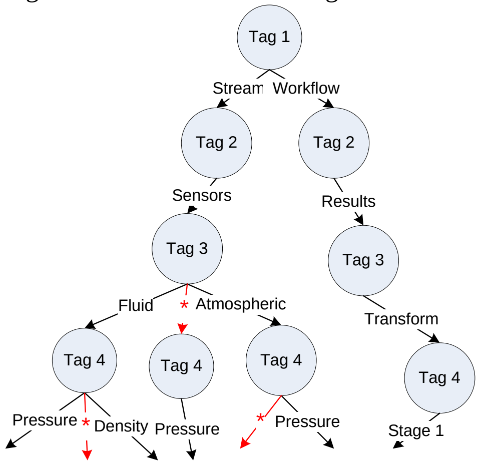


Figure 1: An example subscription tree

3.1.1 Adding and removal of subscription constraints

When processing subscription constraints the tree traversal is 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.

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. 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. 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.

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 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.

3.1.3 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*.

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 within the wildcard's scope.

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 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-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 (encapsulating 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 [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. 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.

3.3 Hashing based

In our 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.

3.3.1 Addition and removal of subscription constraints

In our algorithm, 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.

When a new subscription (depicted in Figure 2.a), needs to be processed, 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 corresponding to this subscription is 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.

```
MANAGESUBSCRIPTIONADDITION(A, consumerDest)
  INITIALIZEWILDCARDCOUNTSARRAY(A_1)
  wcounts = GETWILDCARDCOUNTSARRAY(A_1)

  for i ← 2 to SIZE(A)
    if A_i = *
     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
```

(a)

```
COMPUTEDESTINATIONS(A)
  dest ←-NIL, level ← 1
  wcounts ← GETWILDCARDCOUNTSARRAY(A_1);
  if (wcounts = NIL)
    then return dest
  dest ← GETDESTINATIONFOR(A_1);
  dest ← dest U FINERRECURSION (dest, A, wcounts, A_1, level)


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

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

(b)

Figure 2: Algorithm for adding subscriptions and computing destinations

3.3.2 Computing destinations

To compute destinations ((depicted in Figure 2.b)) 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 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.

3.3.3 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. Please note that 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 also 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.

4 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 will be 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. Readers interested in NaradaBrokering benchmarks in settings involving broker networks are referred to [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 also the number of subscription constraints that are managed by each algorithm. Under these conditions, we report the costs involved in computing destinations for a given stream fragment. These computational costs reveal the suitability of each algorithm for real-time streaming.

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 subscription vary.

We also profiled the tree-based approach for its memory utilization: specifically, we track the number of nodes and edges that are created for different number of attributes as the size of the managed subscriptions varies.

4.1 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 will reveal 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 depending on the quality of the underlying network.

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.

The results, depicted in Figure 3, 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; 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 50-200 milliseconds per hop

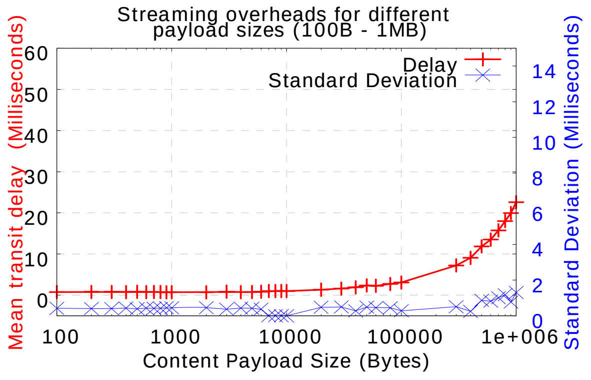


Figure 3: Streaming overheads in cluster settings

4.2 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. We also used a high-resolution timer to report most of our measurements in microseconds.

4.2.1 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.

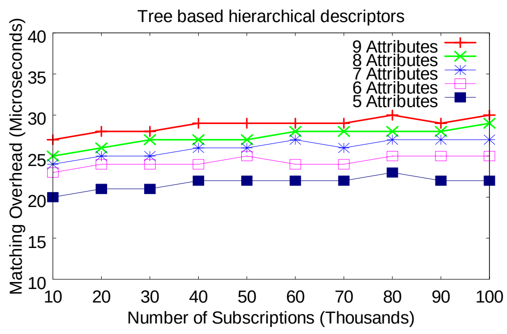


Figure 4: Overheads for tree-based scheme

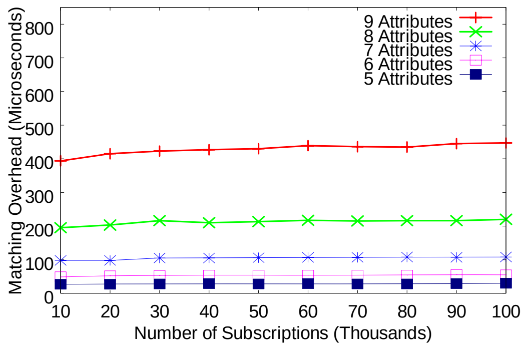


Figure 5: Overheads for the Hashing scheme

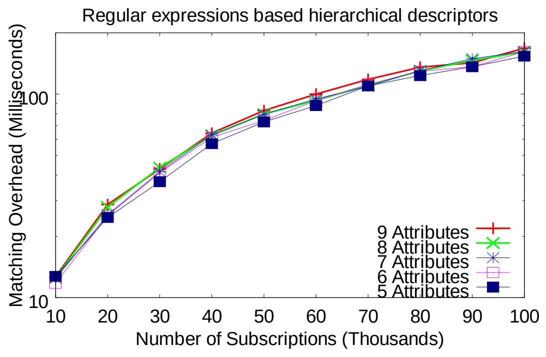


Figure 6: Overheads for Regular expressions

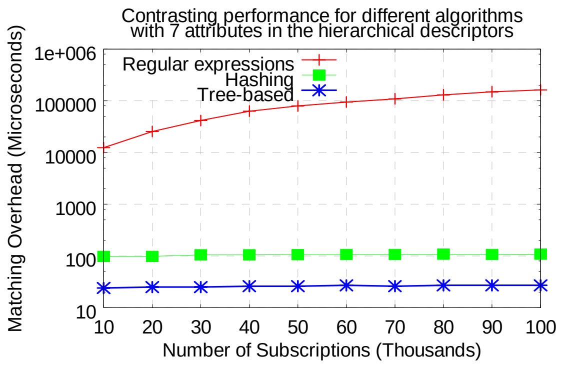


Figure 7: Cumulative overhead comparisons

Figure 4, Figure 5 and Figure 6 depict the overheads for computing destinations in tree-based, hashing and regular expressions scheme respectively. In general, the matching overheads increase as the number of subscriptions and the number of attributes within the subscriptions increase. Given the large number, and random generation, of subscriptions, a wildcard eventually appears for almost every other attribute in a set of related subscriptions (based on the first attribute). This in turn causes the hashing-based scheme – Figure 5 – to approach its worst-case performance wherein the number of sweeps of the Hashtable becomes proportional to the number of specified attributes. In the regular expressions case, Figure 6, the costs (in milliseconds) do not depart significantly from their high base costs as the number of attributes increase.

Figure 7 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.

4.2.2 Space Utilization in the Tree-based scheme

Perhaps the biggest drawback of the tree-based scheme is the memory requirements associated with maintaining the set of subscriptions. Figure 8 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.

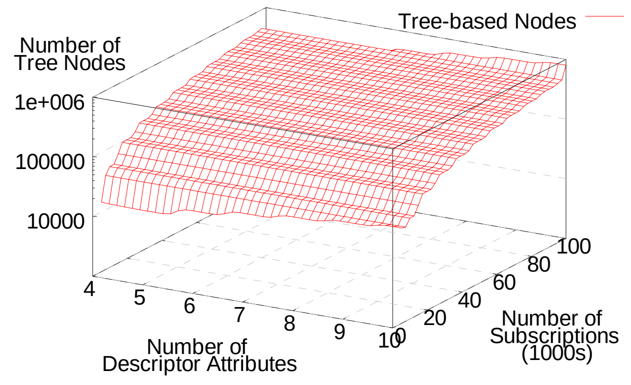
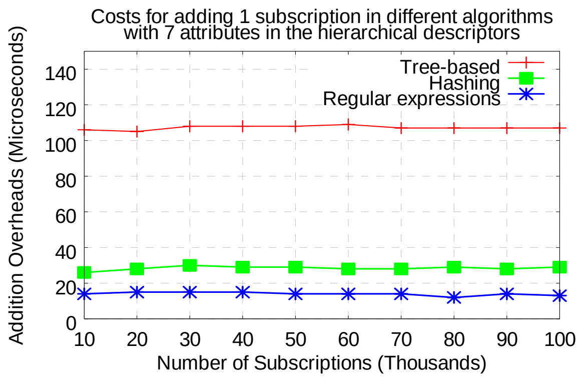


Figure 8: Node allocation costs in tree-based scheme

4.2.3 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 9 and Figure 10 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.



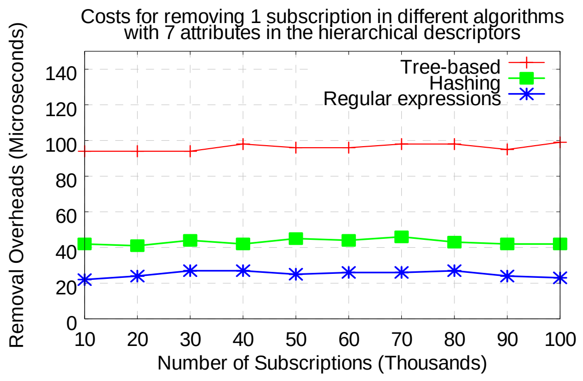


Figure 9: Costs for adding a subscription

Figure 10: Costs for removing a subscription

Standard Deviation

Each point in our graphs (figures 4,5,6,7, 9 and 10) corresponds to the average of a 100 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. One reason we did not plot these standard deviations was because of space constraints.

5 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 very high (Figure 8).

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] to be more in line with the emerging consensus within the Web Services community.

Ref [7] outlines a strategy to convert each subscription in Elvin into a deterministic finite state automaton. This conversion, and the matching solutions, nevertheless can lead to an combinatorial explosion in the number of states for a small number of subscriptions. Systems such as SonicMQ [8] and TIBCO [9] incorporate support for hierarchical "/"-separated topic spaces. However, to the best of our knowledge, they do not seem to include support for implicit wildcard operators.

The Java Message Service (JMS) [10] 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 query evaluation in general tends to be just as compute intensive as the evaluation of regular expressions.

The Event Service [11] 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.

6 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.

In general, all three algorithms coped reasonably well in their ability to cope with the flux in their set of managed subscriptions.

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.

Bibliography

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. Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with Elvin4. In Proceedings AUUG2K, Canberra, Australia, June 2000.
8. SonicMQ: Enterprise Messaging System: www.sonicsoftware.com/
9. P Maheshwari, M Pang: Benchmarking message-oriented middleware: TIB/RV versus SonicMQ. Concurrency - Practice and Experience 17(12): 1507-1526 (2005)
10. M. Happner, R Burridge and R Sharma. Sun Microsystems. Java Message Service Specification. 2000.
11. The Object Management Group (OMG). OMG's CORBA Event Service. Available from http://www.omg.org/