J. Parallel Distrib. Comput. 72 (2012) 450–461

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A complex event routing infrastructure for distributed systems

Hossein Saiedian [a,*], Gabe Wishnie [b]

[a] Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, USA

[b] Bing Infrastructure, Microsoft, Redmond, WA 98052, USA

ARTICLE INFO

Article history:
Received 2 March 2011
Received in revised form
21 October 2011
Accepted 8 November 2011
Available online 18 November 2011

Keywords:
Complex event processing
Distributed systems
Event-based middleware
Cloud computing

ABSTRACT

With the growing number of mega services and cloud computing platforms, industrial organizations are utilizing distributed data centers at increasing rates. Rather than the request/reply model, these centers use an event-based communication model. Traditionally, the event-based middleware and the Complex Event Processing (CEP) engine are viewed as two distinct components within a distributed system's architecture. This division adds additional system complexity and reduces the ability for consuming applications to fully utilize the CEP toolset. This article will address these issues by proposing a novel event-based middleware solution. We introduce Complex Event Routing Infrastructure (CERI), a single event-based infrastructure that serves as an event bus and provides first class integration of CEP. An unstructured peer-to-peer network is exploited to allow for efficient event transmission. To reduce network flooding, superpeers and overlay network partitioning are introduced. Additionally, CERI provides each client node the capability of local complex query evaluation. As a result, applications can offload internal logic to the query evaluation engine in an efficient manner. Finally, as more client nodes and event types are added to the system, the CERI can scale up. Because of these favorable scaling properties, CERI serves as a foundational step in bringing event-based middleware and CEP closer together into a single unified infrastructure component.

1. Introduction

Cloud computing, mega services, and thin clients are becoming mainstream within the industry. Complex applications are moving off the desktop in favor of deployment within the cloud. The cloud computing model is reminiscent of mainframes and dummy terminals. However, our client machines are quite powerful and our mainframes are a cluster of many servers that appear as a single entity to the client as opposed to a centralized server. Whether it be searching the Web, checking email or uploading photos to share with family, the user gets the impression that he or she is interacting with a single service. In reality, there is a controlled chaos behind the scenes among potentially hundreds of servers to perform a single operation. Companies developing these services are constantly driven to add additional servers to accommodate new features, more traffic or lower latencies.

Although the function of these distributed systems is quite diverse, there are common threads that run through all of them. The various components within the system need a mechanism to communicate with one another, and the components perform operations at a staggering rate. While reply/request communication was traditionally used for communication within a distributed system, the industry is starting to adopt event-based communication mechanisms as their primary communication method. There are a few predominant reasons which are causing this shift to occur. Reply/request based communication creates a tight coupling between two components within a system. This is because the initiator has to be aware of the consumer and both require the ability to communicate with one another using the same channel and protocol. Having components tightly coupled complicates upgrades and adds fragility into already complex systems. All of these additional details place unnecessary burdens on the developers of the system and require them to focus on lower level details rather than the business level logic of the application.

1.1. Current event-based middleware limitations

Event-based communication solves the problem of spatial coupling and temporal coupling [3] because the producers of events and their consumers have no knowledge of one another. In other words, the producer and consumer do not directly communicate with one another. Communication within an event-based system is done via middleware, called the event notification service, which serves as a message bus within the system. An event is defined as any observable operation of interest within a system [8]. When a component wants to notify other components

* Corresponding author.
E-mail addresses: saiedian@eecs.ku.edu (H. Saiedian), gabew@microsoft.com (G. Wishnie).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.11.005

within the system that an event has occurred, it will publish a notification using the notification service. In the same right, when a component wants to know when certain events occur within the system it will subscribe for a notification from the notification service. It is then the job of the notification service to ensure that events published by the producers make it to the appropriate consumers. Additionally, the notification service can shield the individual components from much of the complexities of message passing, such as reliable messaging, message ordering and other policies provided by a particular notification service. Centralizing communication in this manner helps provide a single and consistent model for the entire system.

As previously stated, an event is any observable operation within a system. In today's distributed systems, thousands of events take place each second. Because events are being generated so rapidly, it is very difficult for humans to extract much meaning from the event cloud [7]. To illustrate, consider a banking system. The system simply allows users to deposit and withdraw money from their accounts. Using this system, consider the scenario where a user attempts to withdraw money from their account but is notified that it failed. In fact, there are many failed transaction notifications generated within a small timeframe throughout the system. Looking at each transaction, it might not be clear that there is a bigger issue; however, considering all the failure notifications it seems obvious to conclude that there is an infrastructure issue. In fact, at around the same time the bank balance lookup system sent out a notification that it lost connectivity, causing all the failures. Rather than have an engineer look, spending possibly hours, at all the failures and eventually come to the conclusion that all were caused because the balance lookup system lost connectivity, it is better to allow the system to watch for this type of scenario itself and notify the engineer of the exact issue. The ability to look at the system at a higher level of abstraction is not addressed by current publish/subscribe middleware. Rather, a separate complex event processing component is required.

1.2. Current complex event processing limitations

The area of Complex Event Processing (CEP) arose from the need of being able to extract meaningful information from the event cloud created within a distributed system. A typical CEP system consists of two primary components: the query language and the query processor. Instead of allowing only basic subscriptions with filtering, CEP systems allow a query to be submitted which can contain complex filtering and time aggregation. For instance, many CEP systems allow users to submit a query which looks at a group of events within a given time window rather than just individual events. This type of functionality allows aggregation of more frequent events into summary events as well as time based pattern detection. CEP systems also provide a query processor which can receive the notifications being generated within the system and evaluate them against the CEP queries which have been submitted.

Most CEP systems serve as a standard subscriber within the system that listens for all notifications as opposed to the notification service [7]. In other words, there is a dedicated component within the system which serves as the complex query processor. This component instructs the notification service that it wishes to receive all notifications within the system. It will evaluate the notifications against the subscriptions it has received. When a condition within the query is satisfied, the processor will perform various operations such as sending an email or even publishing a new notification within the system to be received by other components. While this centralized architecture allows for easy integration within existing architectures, it also presents scaling issues and does not allow the system to take advantage of certain benefits that would come from combining the message-passing middleware with CEP processing capabilities.

As stated earlier, a typical CEP system is built under the assumption that its query processor will subscribe to all notifications within the system in order to evaluate the queries submitted to it. By centralizing the processor, a bottleneck is inserted into the system as the system generates more notifications. A common solution to this type of scalability issue is to partition the data processed by the query processor. In other words, multiple query processors will now exist and each of them only subscribe to a subset of the notifications. Some of the issues with this solution are how to submit the queries to the correct query processor (the one receiving the notifications being referenced) and how to handle subscriptions that are interested in notifications received by two different query processors. While these issues are solvable, they increase the management complexity of the system each time new notification types or queries are added.

This article presents the architecture of a publish/subscribe middleware which, unlike other publish/subscribe middleware, supports complex queries in a distributed manner. CERI addresses complex queries by presenting a hierarchical event routing transport layer whose goal is to efficiently route events to the appropriate nodes. Each node then has the capability to process these events using CEP semantics. Applications using CERI benefit in two primary ways. First, they can submit complex queries as they would in a standard event subscription. The advantage is that complex queries support a much wider array of functionality, allowing complex logic and filtering to be offloaded to the event processing engine. Second, because all nodes have complex query evaluation capabilities, applications can use the advanced processing capabilities for local operations without suffering the performance loss of communicating with a central processor. A prototype of the above system is provided and tested. The objective of the prototype is to demonstrate that a system with integrated CEP capabilities can scale up to meet the communication needs of a distributed system.

2. Background

Before discussing the current research in the field and the proposed architecture, it is important to first have background knowledge and a basic understanding into publish/subscribe middleware. This section provides this additional context by first introducing some of the basic architectures that are often used when developing distributed publish/subscribe middleware and then describing the core concepts that differentiate complex event processing from standard event subscription models. Many of the concepts utilized within CERI's overlay network build upon this body of work.

2.1. Architectural concepts

As distributed systems grow, they often evolve in terms of hardware, network and architecture. If all applications that make up a distributed system had to be aware of these changes, the distributed system as a whole would grow very complex and instable. Middleware is often used to allow for the natural evolution of a distributed system while reducing the impact of such changes to all components. Middleware is a logical layer that sits between the consuming application and the physical operating system and hardware [13]. Using middleware localizes the needed modifications based on physical changes into a single layer of the distributed system, allowing the consumers to remain unchanged.

When a distributed system uses an event-based model for communication, the application generating the notifications (producers) and the applications consuming the notifications (consumers)

Table 1
An overview of the common relationships among events.

Time	Ordering of events based on a clock. e.g., A user deposits $500, then later withdraws $100.
Cause	Dictates that one event must have occurred in order for another to have occurred. e.g., A user withdrew $500 causing an overdraft to occur.
Aggregation	Contains the set of events that make up the higher-level event. e.g., A user gets an overdraft notice because $500 was deposited, $100 was withdrawn and $500 was withdrawn.

are not aware of one another. Instead, a consumer simply lets the publish/subscribe middleware know that it wishes to receive certain notifications. When the consumer publishes the notifications that match a subscription the middleware ensures that it goes to the appropriate consumers. While this sounds straightforward, the challenge comes when considering the scale at which many of the enterprise level distributed systems operate. In other words, there are thousands of producers and consumers generating notifications at extremely high rates. Fortunately there are a few fundamental architectural concepts that have been developed for such circumstances.

The first major distributed systems were built for the purpose of file sharing. Initially, they were implemented with a centralized index node that stored the mapping of files to nodes. When a node wanted to locate a file, it first contacted the central index to locate which nodes had the file being requested. Once located, it would go directly to those nodes to download the file. In fact, the once infamous Napster file sharing network was implemented with the above design. Because all nodes on the network first had to contact the centralized index node before downloading a file, the centralized index node served as the bottleneck to the system. Realizing the limitations of a centralized approach, other system architectures soon arose attempting to decentralize the lookup process [11]. The next generation of distributed systems were designed in such a way that nodes simply contacted one another for file locations rather than a centralized index node. Because of the decentralized nature of these systems and the direct communication among nodes, this type of system is referred to as having a peer-to-peer architecture [13]. Just as file sharing distributed systems started out utilizing a centralized architecture and later evolved into decentralized peer-to-peer architectures, publish/subscribe middleware also followed the same progression [5].

Peer-to-peer architectures fall into two primary categories: structured and unstructured. The primary difference between the two is the method in which the network is constructed. Structured peer-to-peer systems are constructed in a deterministic way, while unstructured are not [13]. Currently the majority of proposed publish/subscribe middleware systems utilize a structured peer-to-peer network. Specifically, most publish/subscribe middleware systems utilize some form of a distributed hash table (DHT). As implied by the name, a DHT uses some form of hashing to distribute data and workload among the nodes in a system. Within a DHT based publish/subscribe middleware, peers share the task of storing subscriptions, matching subscriptions and delivering notifications. The challenge of using DHT based architecture is distributing load in a uniform manner across all nodes and performing efficient lookups [16]. Some approaches used to solve the previously mentioned issues of utilizing a DHT based architecture within publish/subscribe middleware will be addressed in more detail in Section 3.

2.2. Complex event processing

As the popularity of event-based communication grew within the software community, so did the realization that as a distributed system grows larger it is extremely difficult to get a good picture of the behavior of the system at any one point in time. Consider again the banking application mentioned earlier. Within this system at any point in time there can be thousands of account holders making deposits, withdrawals and transfers, each of which lead to the generation of one or more events. In addition to the user triggered events, the system itself is producing events to notify of things such as network health, server health and component level failures. The sheer volume of events within a distributed system at any point in time make it all but impossible for a human to make any kind of meaningful correlation without help. The field of complex event processing (CEP) was developed to help in the situation described previously. Essentially it strives to give meaning to the meaningless cloud of events within a system.

Traditionally, publish/subscribe middleware systems adhere to one of three types of subscription schemas: topic-based, content-based and type-based [3]. Topic-based is the oldest and simplest subscription schema. Within this subscription schema each event is assigned to a particular topic. The subscribers can request which topics they wish to receive. Topics can be hierarchical in order to provide some flexibility, but even so the filtering is limited. The content-based subscription schema provides additional filtering capabilities over topic-based filters. Systems that allow for content-based subscription filtering enable events to be filtered by each of the attributes or properties associated with the event. Allowing the additional granularity in filtering reduces the number of unwanted events a subscriber receives. For instance, consider a component within the example banking system that wishes to receive all deposit events with an amount of $500 or greater. In a topic-based system the subscriber would receive all deposit events regardless of amount and would then have to further filter out the ones that meet the additional amount criteria. However, in a content-based system this logic could be done in the subscription itself, meaning that the logic in the subscribing component is simplified as the extra filtering is no longer needed. Finally, the type-based subscription schema provides much of the functionality of the content-based schema with the addition of a type system being imposed on the events. By utilizing a type system with events, a hierarchical and even polymorphic relationship can be used in filtering, much like what is present within a topic-based schema. While all of the previously mentioned subscription schemas are useful, they have the restriction of being able to only consider a single event at a time.

In traditional publish/subscribe middleware event subscriptions are typically limited to evaluating against a single event at a time. While this is useful for receiving events that match certain criteria, it still requires that subscriber components perform any aggregation and correlation their business logic requires. Additionally, looking at raw events in this manner provides little benefit to humans monitoring large scale distributed systems. For this reason, CEP systems go beyond the previously mentioned filtering capabilities and allow relationships between events to be represented using advanced pattern matching languages. Specifically, the most common types of relationships considered to exist between events are time, cause and aggregation [7]. Table 1 provides examples of what each of these relationships may look like within the example banking system.

When observing relationships of this nature within the system, a new entity is produced called a complex event. A complex event

is simply an aggregate of one or more events that caused it to occur. In other words, a complex event contains the events which caused it [7]. Complex events provide much more information than individual events. When connecting the causality of individual events it is necessary that the application developer or user have some understanding into the problem domain. However, complex events contain the events that caused it to occur, making it obvious as to what series of events lead to the current state.

3. Existing event-based middleware

The field of event-based middleware has been around for almost 20 years. Although this may not seem young in terms of the computer industry, it remains relatively immature with the vast majority of research completed within the last five to seven years. The recent influx of research in the field can be contributed to the growing trend of businesses to build cloud-based applications rather than traditional client-based applications. This in turn increases the number of distributed systems being developed which, as demonstrated previously, often rely on event-based communication. As the amount of research increased, trends arose among the various systems. Eugster et al. [3], provides a good dissection of the various pieces that form an event-based middleware system. Rather than provide that type of generalization, this section attempts to show the evolution through a discussion of some of the specific systems which have been proposed throughout the years. In addition, the limitations of these existing systems will also be noted to demonstrate the motivation for constructing the system proposed in this article.

The systems being discussed in the remainder of this section have been broken down into two major categories based on the type of overlay network they implement: hybrid and DHT. The final category in this section diverges from this model to discuss the research done which directly relates to CEP.

3.1. Hybrid overlay networks

The first implementations of event-based middleware did not have to deal with the scale present in today's distributed systems. For this reason, they were built with a centralized server acting as the broker between the various clients who published and consumed events. One such system that illustrates this architecture is Yeast [5]. As alluded to earlier, Yeast uses a centralized server to manage all clients within the system. Clients would then send what were termed specifications, or subscriptions, to the server in order to describe the pattern of events which it was interested in receiving. With the growing popularity of the Internet and the realization that the centralized server approach would no longer scale, the focus turned toward solving the problem of scalability.

In 2001, Carzaniga et al. [2] presented SIENA, which would serve as the basis for much of the future research in the field. SIENA was unique in the fact that it was first to provide an expressive subscription language while also scaling to meet the needs of larger networks. It did so by distributing the evaluation of subscriptions and delivery of notifications throughout all client nodes within the network rather than a centralized location. Specifically, SIENA supported two primary types of network topologies: hierarchical and peer-to-peer. To combat the performance bottleneck broadcasting events to all nodes within the network, SIENA broadcast the subscriptions (subscription forwarding) and, optionally, the events published by each node (advertisement forwarding). The decision to broadcast in this manner was based on the fact that there are typically far less subscriptions than notifications. With the communal knowledge of which nodes require which notifications, the routing path can be optimized for efficient delivery.

While SIENA allowed for expressive filtering and basic pattern matching, it did not implement a complete pattern matching language. In fact, the assumption was made that any necessary matching beyond the capabilities of the language would be done externally. In other words, if a node needs to construct a more complex subscription than that allowed, it would have to instead decompose the complex query into simpler ones that SIENA could understand and perform the more advanced filtering locally.

At around the same SIENA was developed, Pietzuch and Bacon [9] presented a more traditional event-based middleware system dubbed Hermes. As with SIENA, Hermes distributed subscription evaluation and event propagation throughout the nodes on the network. However, Hermes provided a dedicated function, called an event broker, which was responsible for performing these functions. Within Hermes' network the event brokers were interconnected with client nodes, called event clients, which are connected to the brokers. To extend the event filtering capabilities beyond that provided by type-based subscriptions, Hermes also supported basic content-based filtering. The content-based filtering was performed by the brokers as close to the source as possible to reduce unnecessary traffic. Similar to SIENA, Hermes's filtering capabilities were tightly bound to the network topology. For this reason, neither made ideal candidates for integrated CEP capabilities as they would require extensive modifications.

3.2. DHT overlay networks

In addition to the various types of hybrid overlay networks developed for use within event-based middleware, the use of a DHT overlay network is often exploited due to the inherent ability to provide efficient lookups across a distributed network. One of the first such systems is called Chord [12]. Although Chord is not an event-based middleware in its own right, it serves as the basis for many such systems. For this reason, it warrants a brief examination.

Chord was constructed with only a single operation in mind. Given a key, it can map it to the node which contains the associated data. Although seemingly simple, the problem is quickly complicated by the fact that nodes can leave and join the network at any time. When a node joins the network it is assigned an identifier based off its IP address. The nodes are organized in a ring based upon the identifier they have been assigned. The node is then assigned a set of keys through the use of a consistent hashing algorithm. When a node joins the network it is given some of the keys of its successor within the ring. In the same way, when a node leaves the network, its keys are assigned to its successor within the ring. Assigning keys in such a manner provides a balanced distribution across all nodes in the ring, and also provides for efficient entry and exit from the node ring.

Due to the previously discussed characteristics of Chord, various event-based middleware systems are based upon this overlay network. Two such systems are detailed by Terpstra et al. [14] and Yang and Hu [16]. Terpstra et al. [14] describe a system in which the task of delivering a notification and evaluating a subscription is distributed among all the nodes within the ring. Unfortunately, there is no specific subscription schema implemented within their system, so it is unknown as to how it will perform. Additionally, it relies on broadcasting for the distribution of subscriptions. Although there are often fewer subscriptions within a system than notifications, it could lead to a bottleneck.

Yang and Hu [16] demonstrate that a content-based subscription schema can be constructed atop a Chord overlay network. They accomplish this by modeling the entire system's schema as a multi-dimensional space. Each dimension within this space represents an attribute that comprises the event. By describing the space

in such a manner, the event can then be thought of as a point in the space and the subscription is defined as a hypercuboid. A match is made when the point represented by the event is within the hypercuboid represented by the subscription. Unfortunately, because a new dimension needs to be created for each event attribute, the model breaks down as the diversity of events within the system increases.

In addition to the Chord overlay network which supports a single lookup operation, another foundational design was developed by Ratnasamy et al. [11] to provide similar functionality. The content-addressable network (CAN) attempts to solve the same issue of a scalable distributed lookup operation but in a different manner than that used by Chord. Specifically, a CAN maps a virtual coordinate among all nodes within the overlay network. Each node is responsible for the coordinate space it is assigned. When a key/value pair is added to the system, the key is first hashed using a uniform hash function and then mapped to a point within the coordinate space. In the same way, when data needs to be located, the hash of the key is computed which locates the coordinate. Thus, by knowing the coordinate, the node storing the value is also known.

Just as with Chord, many event-based middleware systems are based on a CAN. Each typically focuses on a single shortcoming of the initial implementation in an attempt to overcome the limitation. For instance, Gupta et al. [4] present Meghdoot, which focuses on evenly load balancing among all the peers in the network. The limitation they focused on was that of the uniformity of the hash function to distribute data among all nodes. Specifically, they proposed that real world data is often skewed, causing the hash function to break down. To solve this problem, they allow nodes to share knowledge of their current load and, if needed, rebalance to reduce the potential bottleneck. In addition to the Meghdoot system, Bharambe et al. [1] developed Mercury, which adds support for multi-range queries within a DHT overlay network. While the addition of range operators helps improve the flexibility of the subscription language, causal relationships and other advanced operators required by CEP still cannot be represented, causing the model to break down.

Overall, DHT based overlay networks present interesting techniques at efficiently balancing data, balancing load and performing efficient lookups among a large distributed network. The problem lies in the fundamental motivator which was used to design these types of networks. When built, they were optimized for a single operation: a value can be located efficiently given a key. While there has been research done to improve the available operations provided by subscription languages, each additional operator has to be carefully mapped into how the network is built. Due to the number and complexity of operators supported by complex query languages, it is impractical to attempt to support them natively within a DHT based overlay network. For this reason, a DHT was not utilized by the system proposed in this article.

3.3. CEP frameworks

As evident by the amount of research available, the field of CEP is much younger and less investigated than that of general event-based middleware systems. Typically the research related to CEP does not consider the act of event routing, just as the previously mentioned systems do not consider support for CEP. For instance, Pietzuch et al. [10] propose a composite event (CE) detection framework which allows events to be composed into complex events rather than subscriptions only considering single events. The problem is that the proposed framework merely acts as a simple subscriber and publisher within the existing event infrastructure. In other words, the CE detection component subscribes to all events or a specific set of events using the capabilities provided by the current event infrastructure. As complex events are created, they are published back to the existing event infrastructure for distribution. The same type of implementation is utilized by Wu et al. [15] when presenting SASE. Rather than focus on an end-to-end implementation, the research instead focuses on the challenges presented by CEP.

While implementing the CE or CEP framework as a pluggable component provides great flexibility for integration into existing systems, it falls short in two primary areas. First, the scalability model of the systems is different, which adds complexity into the overall distributed system. In other words, a lot of focus is given to scalability within the field of event-based middleware. However, scalability is often overlooked when presenting a framework for CEP. Ideally, the CEP system can take advantage of the extensive research in this problem space done within the event-based middleware field. Unfortunately, if the CEP engine acts merely as a subscriber to events, it cannot. Second, the application developers must learn two systems and often cannot easily take advantage of the capabilities of the CEP toolset. Consider if an application wants to only receive a complex event which has child events that meet a specific criteria. Because the event infrastructure itself has no knowledge of complex events it is not possible to construct a query to meet the needs of the application. For these reasons, the system proposed in this article considers CEP capabilities as a primary function within the systems and merges the message passing and CEP processing into a single infrastructure.

4. Proposed event-based middleware

This section introduces the architecture of the complex event routing infrastructure (CERI). As described in the following section, CERI utilizes several concepts from previous bodies of work for scaling and routing optimizations but does so with the focus on integrated CEP. CERI consists of two primary components: the event router and the local event router. The role and high level design of each of these components are discussed in detail in the following subsections.

4.1. The event router

The event router's primary responsibility is to facilitate the process of events published by an application reaching the applications that require them. Because of the wide range of operators available to event matching languages used for CEP, it is difficult to effectively filter out all candidate events before they reach the client. For this reason, there are two levels of filtering within CERI. The first level occurs at the event router level while the second occurs on the clients themselves as part of query evaluation. Due to the tiered filtering, it is inevitable that more events will need to be routed than are actually needed for query evaluation. Typically publish/subscribe systems attempt to hash the subscriptions and events over a structured peer-to-peer network [4]. While this provides the needed functionality to support basic operators in the subscription, it is difficult to perform range queries and other advanced operators efficiently. As these operators are part of event pattern matching languages used for CEP, it is necessary that they are efficiently supported within CERI. For this reason, CERI's emphasis is on efficient event routing, allowing for query evaluation on all clients. With this in mind, CERI diverges from most publish/subscribe middleware systems and uses an unstructured peer-to-peer network rather than a structured one.

Unstructured peer-to-peer networks differ from structured peer-to-peer networks primarily in how the overlay network is organized [13]. While structured peer-to-peer networks are

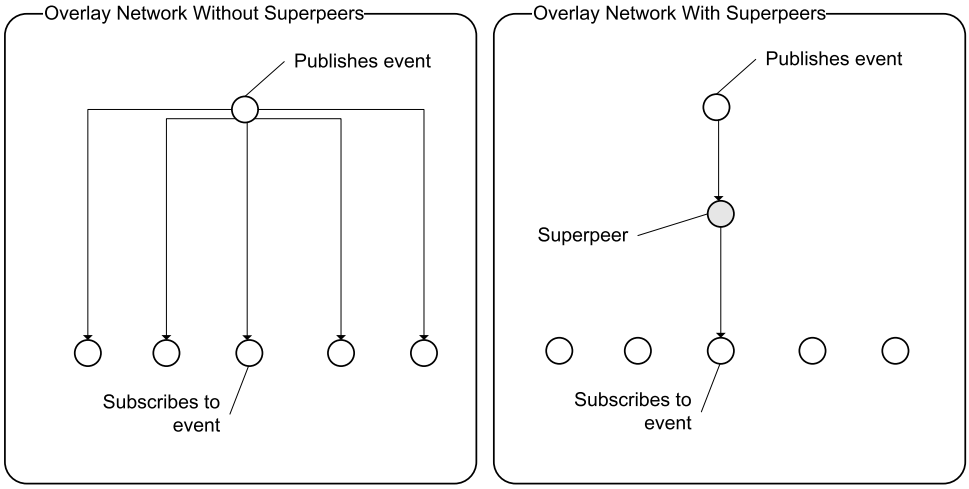

Fig. 1. The benefits of superpeers within an unstructured peer-to-peer network. Without superpeers, an event must be broadcast to all other nodes regardless of whether it is needed to. With the use of a superpeer, the events required by each node are indexed. When an event is published, it first goes to the superpeer which will only broadcast to the appropriate nodes.

organized in a deterministic way which can be exploited by the distributed system built on top of it, unstructured peer-to-peer networks are built in a much more random manner. Although the random construction of an unstructured peer-to-peer network often helps nodes join and leave in an efficient manner, data lookup complexity is increased. To combat the issue of inefficient data lookup, CERI uses superpeers to aid in event routing [6]. Fig. 1 illustrates the benefit of using superpeers to aid in event routing.

In CERI, the event router acts as the superpeer within the system. When a node joins the network it will connect to an event router. As a node creates subscriptions, the event router will index the type of events required by all the subscriptions on that particular node. The event router will then perform the first level of filtering previously mentioned and only route the types of events actually needed by the node for subscription evaluation. Because large distributed systems often contain a large number of event types, performing the initial filtering on the event router reduces the footprint of CERI on each individual node.

For scalability purposes, there are multiple event routers within the system. By allowing multiple event routers, the system can be scaled out rather than up as the peer-to-peer network grows. For simplicity, rather than allow elections to determine which peer among a pool of candidates will serve as an event router, the event router role is a dedicated function within the system. In other words, the machines acting as event routers will not be running client applications. Making the event router a dedicated function was done for two reasons. First, to make certain that an event router is not executing client applications ensures that a poorly written application cannot negatively impact a large set of nodes relying on the given event router node. In other words, if a client running on the event router were to consume the majority of system resources causing the event router capabilities to suffer, all additional nodes would suffer. Second, by providing a dedicated function, a higher density of nodes to event routers will exist, reducing the cross communication that happens among the event routers. The separation of function was primarily completed due to the large amount of event filtering that will be taking place on the event routers. If the event router is the only task running on a particular node, then the risk of starvation to any client application running on the same node is not a threat.

To aid child nodes in selecting a potentially optimal event router, the event router publishes a number of metrics to give an indication of its current performance and load. These metrics include incoming events per second (EPS), outgoing EPS, subscription count and client count. Using these metrics, a child node can get an indication of not only if it would like to connect to the event router in the first place, but also if it would like to switch to another less congested event router once connected. By allowing performance interrogation of this nature by the child node itself, starvation of a particular node is less likely to occur as it can decide at any time if it would like to relocate. Load balancing among the event routers themselves is another option to alleviate this type of congestion. Yeung Cheung and Jacobsen [17] provide a comprehensive investigation into load balancing techniques within a distributed content-based system called PEER. However, these methods were not considered within this design.

As a distributed system scales out, it is necessary to add additional event routers. When more event routers are added, network flooding again becomes an issue. For this reason, event routers are grouped into zones. Zones are a logical grouping of a set of event routers. This grouping can be completed methodically, such as grouping machines with similar functions into the same zone, or it can be done randomly. Zones help reduce network flooding by distributing the event type filtering that occurs on the event router among the entire zone. This is accomplished by allowing event routers within a zone to exchange the list of event types their children require. In other words, CERI performs subscription forwarding within zones as an optimization technique [2]. Now, when a node publishes an event, its event router will know if any other event routers within the zone have children which require the event. With this knowledge, the event router will only transmit the event to the appropriate subset of event routers as opposed to all event routers just to be potentially discarded by most. Fig. 2 demonstrates these differences.

With the formation of zones the overlay network is effectively partitioned. Applications that tend to depend on one another are grouped within the same zone as inter zone event transmission is much faster. However, because publish/subscribe systems decouple publishers from subscribers, this is not a requirement. For this reason, the role of a gateway event router is introduced. The gateway event router is effectively a superpeer among the event routers. It performs the same function as any other event router with the difference that it will also be the gateway between its zone and all others within the overlay network. An election is held within each zone among all the event routers to determine which one acts as the gateway event router. Once the winner is decided, all event routers are notified as to who is the acting gateway event router.

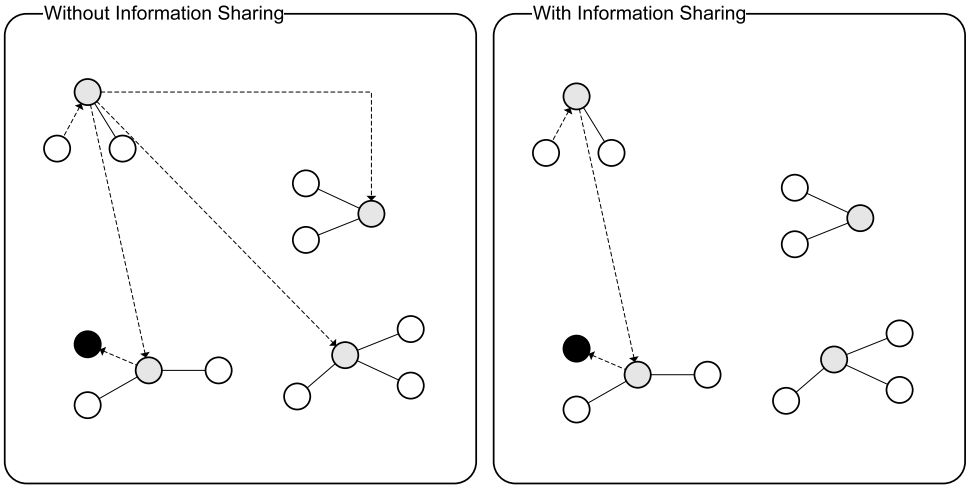

Fig. 2. The benefits of event router information sharing within zones. On the left is a zone in which the event routers (the gray nodes) do not share the events required by their child nodes (the white nodes) with other event routers. Because of this configuration, when an event is published, it must go to all event routers even through it only needs to go to the event router of the nodes that subscribe to the event (the black node). On the right the same scenario demonstrates the reduction of network communication when publishing an event while information is shared within the zone.

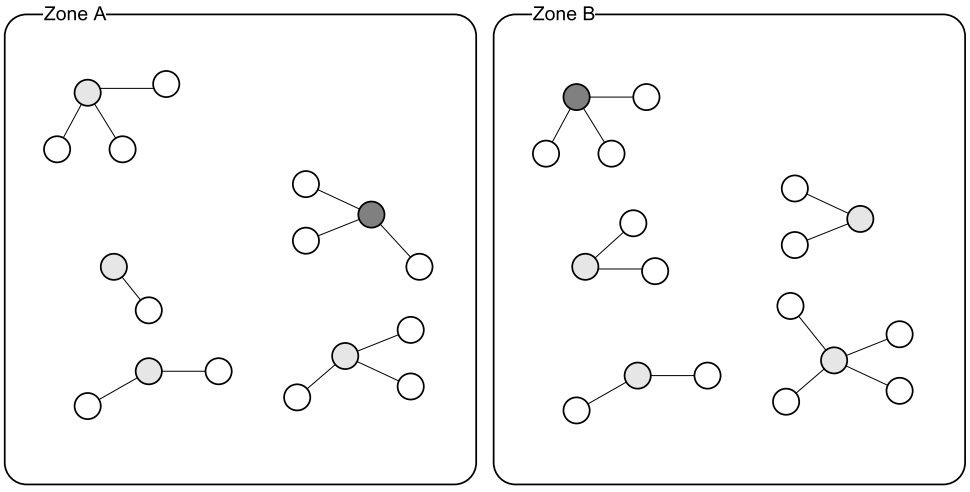

Fig. 3. CERI's overlay network. An example overlay network that contains two zones. Each zone has a single gateway event router (dark gray nodes), multiple event routers (light gray nodes), and potentially many child nodes (white nodes).

The gateway event router has the extra responsibility of acting as the mouthpiece of its zone. As a result, all events that originate within a zone are always published to the current acting gateway event router. Once a gateway event router receives an event from its zone, it publishes it to the gateway event routers in all other zones. In the same right, the gateway event router receives all events that originate from external zones. Because the gateway event router has the knowledge of what events each other event router requires, the events are then directly distributed. The organization of event routers and peers within the overlay network is shown in Fig. 3.

The event publication process is composed of six distinct steps as described below:

1. An event is sent from a client machine to the event router.
2. The event router forwards the event to all appropriate event routers and the gateway event router within its zone.
3. The gateway event router distributes the event to all other gateway event routers.
4. Within each zone the gateway event router forwards the event to all appropriate event routers.
5. The event router sends the event to all appropriate client machines.
6. The client machine receives the event and evaluates it against its queries.

For clarity, the described event publication process is illustrated in Fig. 4.

Using the above communication model, an event will be routed through at most four event routers for inter zone communication. To achieve this upper bound, gateway event routers must have knowledge of all zones within the system. If the system's scale surpasses practicality, gateway event routers can be modified to only contain a partial view of the zones. However, such a configuration increases the potential number of hubs when routing events between zones since each event would have to be published by multiple gateway event routers rather than only one. For inter-zone communication, an event will be routed through at most two event routers. This is possible because all event routers within a zone exchange the list of events required.

4.2. The local event router

In the same way the event router has the responsibility of ensuring nodes within the overlay network receive the appropriate events, the local event router's responsibility is to ensure that

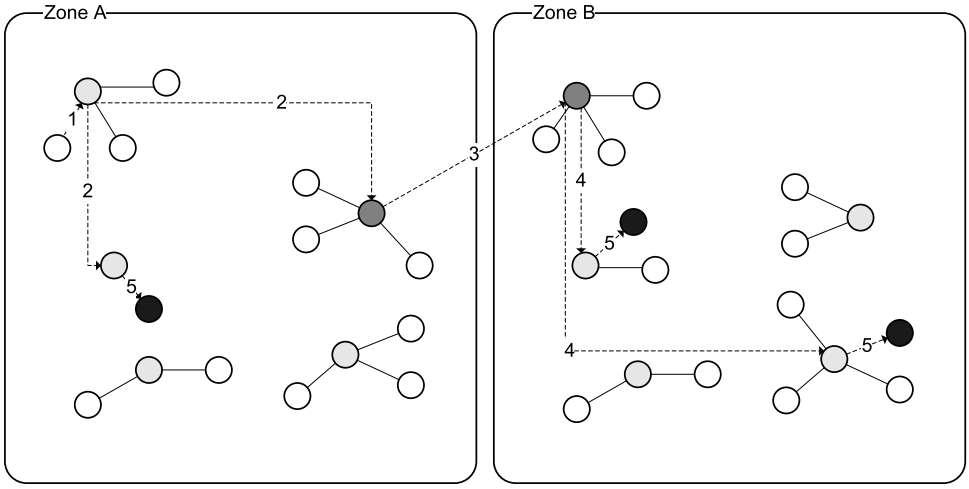

Fig. 4. The event publication process. The steps indicated correspond to the process outlined above. Note that step 6 is not shown as this happens on each client and does not require an additional transmission.

Table 2
The functionality provided by the CERI API.

RegisterForEvents	Allows client applications to register for a set of events
UnregisterForEvents	Allows client applications to unregister for a set of events
UnregisterAllEvents	Allows client applications to no longer receive events
PublishEvent	Allows client applications to publish an event
AddListener	Allows client applications to receive a callback when a query match occurs

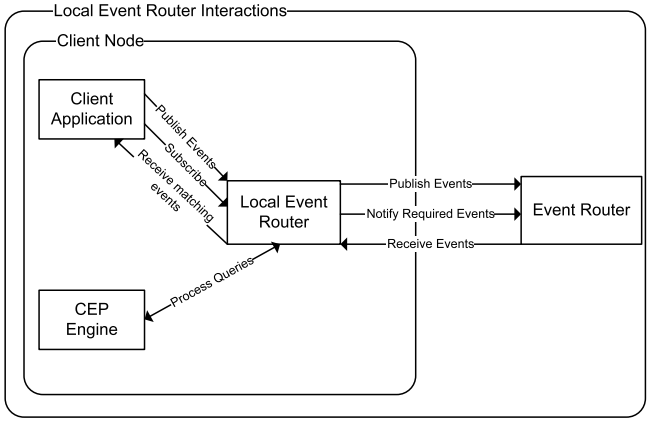

Fig. 5. The local event router interactions. The local event router serves as the facilitator on the client node between the client application, the CEP engine and the event routing infrastructure.

once events reach the node, they are routed to the appropriate application for evaluation against the complex query. In addition to event routing capabilities, the local event router has the extra responsibility of facilitating the publication process with the client applications and managing the connection of the node to the overlay network. Fig. 5 demonstrates the interactions between all of CERI's components.

When a node wishes to join the network, the local event router will begin the process of locating an appropriate event router to make a connection with. Once connected to an event router, the node is ready to create subscriptions and begin publishing notifications. To aid application developers in both of these processes, a simple application programmer interface (API) is provided as a library. Intentionally, the library contains minimal functionality, detailed in Table 2.

When the local event router receives an event to publish it adds it into a publication queue. When dequeued, the local event router attempts to publish the event to the event router. If the publication fails, the following actions are taken. First, the local event router will attempt to make a configurable amount of retries to the known event router. If still unsuccessful, the node and/or event router are most likely having connectivity issues. The local event router will next attempt to locate another event router to which to connect. If successful, the event will be published as expected. However, if still unsuccessful, the local event router will continue to retry while allowing new events to be queued.

In the same way a client application can publish an event, it can also create a subscription. The subscription itself is evaluated locally. By locally evaluating a subscription, CERI allows applications to efficiently offload internal logic to the CEP engine without incurring the performance penalty of having to transmit the event to an external machine. However, in order to perform the first level of type based filtering, the types required for evaluation are extracted. Once the types have been extracted, they are processed for uniqueness. In other words, if the types required have already been requested for the given node for other subscriptions, no action is taken. Otherwise, if new types of events are now required, the event router must be notified in order to begin routing the additional events to the node. In addition to sending required types to the event router when a new subscription is added, the event router also has the ability to request the node send all event types it requires. This is primarily used when a node switches between event routers. The first thing the event router does is request the full list of event types the node wishes to receive.

In addition to facilitating the publication and subscription processes, the local event router interacts with the CEP engine. The local event router can interface with any CEP engine and serve as an adapter to transform the events in a format which can be understood by the engine. By not restricting the engine, a CEP engine can be selected to best support the needs of the organization. As noted earlier, evaluation of complex queries takes place on the client nodes themselves. The benefit of localizing this process is that not only can the system scale as nodes are evaluating their own subscriptions, but applications

can also efficiently take advantage of the functionality provided by the CEP engine. For instance, a client application need not only generate events for distribution but rather generate events for their own consumption. Because complex pattern matching languages support rich functionality and the subscriptions are evaluated locally, applications can better utilize the infrastructure for handling error prone logic that once would have been implemented from scratch. By offloading logic in this manner, additional benefit is added to the consuming applications.

5. Experiments and analysis

In order to prove that the theoretical design can be implemented in an efficient manner, a prototype was constructed and experimented on. This section begins by describing the technology stack used by the prototype. Next, the implementation level design details are covered. This includes some of the lower level architectural choices and the reasoning that went into making them. Finally, the results of the experimentation are presented. Because CERI is not responsible for processing the complex queries, the experimentation primarily focused on event routing speed.

5.1. Prototype design

One of the difficult tasks when constructing any system is deciding which technologies best balance the various requirements. With the requirements for CERI in mind, Table 3 enumerates the category and chosen technology used when constructing the prototype.

As shown in Table 3, the prototype primarily utilizes a Microsoft technology stack. This translates into Windows Server 2008 serving as the platform and.NET 3.5 as the development platform. Additionally, Windows Communication Foundation (WCF) is used when performing both inter process and network communication. WCF was selected because it provided a simple way to modify both the protocol and the channel being used for communication in order to find the preferred combination. Finally, NEsper was selected as the CEP engine used to perform complex query evaluation in the prototype. NEsper was primarily selected due to the flexibility it provides. It supports a very simple hosting model which is exploited in the prototype.

Events within CERI are represented with plain-old.NET objects. The only restriction placed on the event itself is that it must be serializable. This is necessary because events must have the ability to be transmitted over the network. For performance reasons events are not deserialized until they reach their destination. To allow the first of type based filtering to occur on event routers, the type is transmitted separate from the main payload of the event. The sequence of events that occurs when an event is published by an application is shown in Fig. 6. When the event reaches an appropriate node for query evaluation it is deserialized, converted as necessary for the CEP engine and evaluated against all queries.

5.2. Experiments

In order to gage the effectiveness of the overlay network implemented in CERI, simulations were performed on the prototype described in the previous section. Because CERI's primary focus is on the transmission of events in a way which supports CEP, the focus of the simulations is on the various transmission test dimensions rather than query processing. It is understood that whatever CEP engine is chosen for usage in conjunction with CERI will take some non-zero time for processing the queries; however, for the purposes of this article, this processing time is not considered. This is primarily due to the volatile nature of measuring the time to process a CEP query. The processing time can fluctuate based on any number of factors including, but not limited to, the number of events being considered in a query, the time window being considered and the time complexity of the operations being utilized. For this reason, only the transmission time is considered to provide the most stable set of metrics possible.

Table 3
The prototype's technology stack.

Operating system	Windows 2008 enterprise edition
Development platform	.NET framework v. 3.5
Networking platform	Windows communication foundation
Logging library	Log4net
CEP engine	NEsper

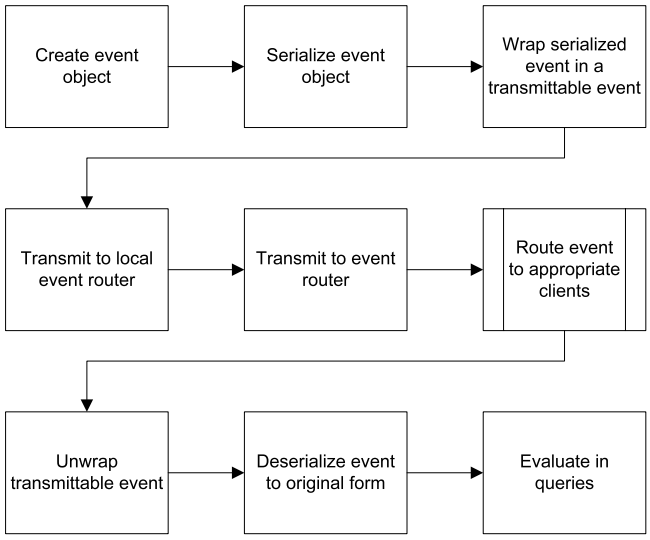

Fig. 6. An event's transmission process. The various steps an event undergoes when published on a node, transmitted through CERI and arriving on a subscriber node.

The simulation process was intentionally kept simple and straightforward as to not introduce false latencies into the overlay network. To obtain measurements, the prototype has been instrumented using Windows Performance Counters. Furthermore, all countermeasures implemented to reduce saturation of the event router by a single client have been disabled. Because the simulations are in a controlled environment, they were deemed unnecessary as they would only negatively impact the results and artificially throttle the performance metrics. Lastly, a test client was developed and instrumented in the same manner as described above. The test client serves as the means to control which events are published and subscribed to within the system under test.

The first simulation was targeted at determining maximum throughput of the system at different payload sizes. Essentially, the aim is to determine the effects of payload on throughput. For this test only two clients were added to the system, with a single event being transmitted between them. One of the clients served as the publisher and the other as the subscriber. It is thought that payload size will have a significant impact on throughput. The reason for this hypothesis is two-fold. First, the event must be transmitted. The more data required to be sent among nodes per event, the more saturated the components become, causing throughput to decline. Second, before an event can be transmitted, it must first undergo a serialization/deserialization process. The time required for this step is directly impacted by the amount of data contained in the event.

The test was performed with an event payload size of 1, 100 and 500 KB. For a single event, the upper bound of 500 KB is reasonable. If a distributed system expects to be sending larger events often, it is recommended that an event-based middleware be optimized for

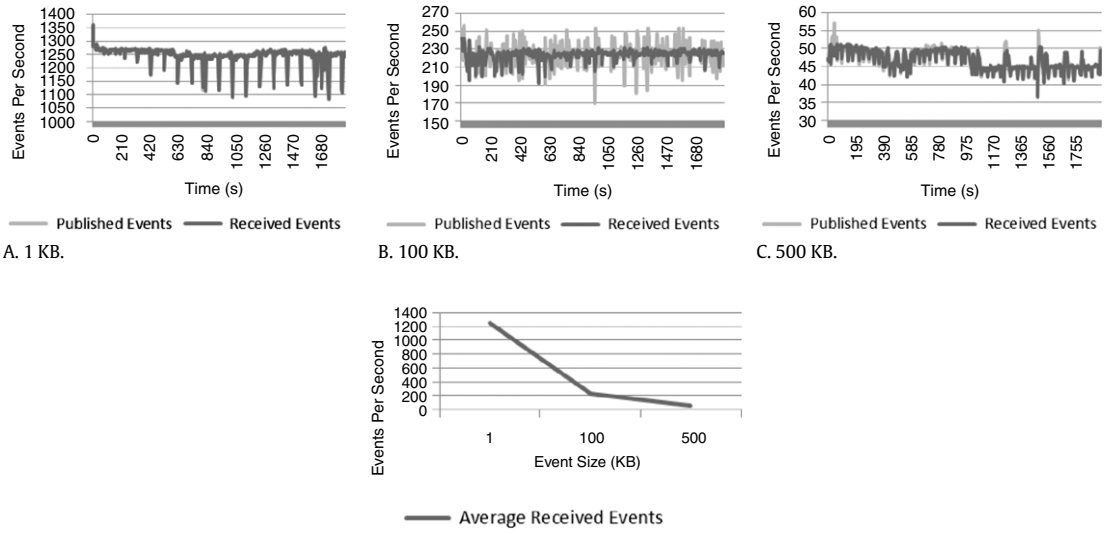

Fig. 7. Event size simulation results. Subfigures A–C show the raw throughput of the three event sizes tested in kilobytes for the duration of the experiment. There is a strong negative correlation between event size and throughput. Subfigure D shows the average throughput of each event size for the same period.

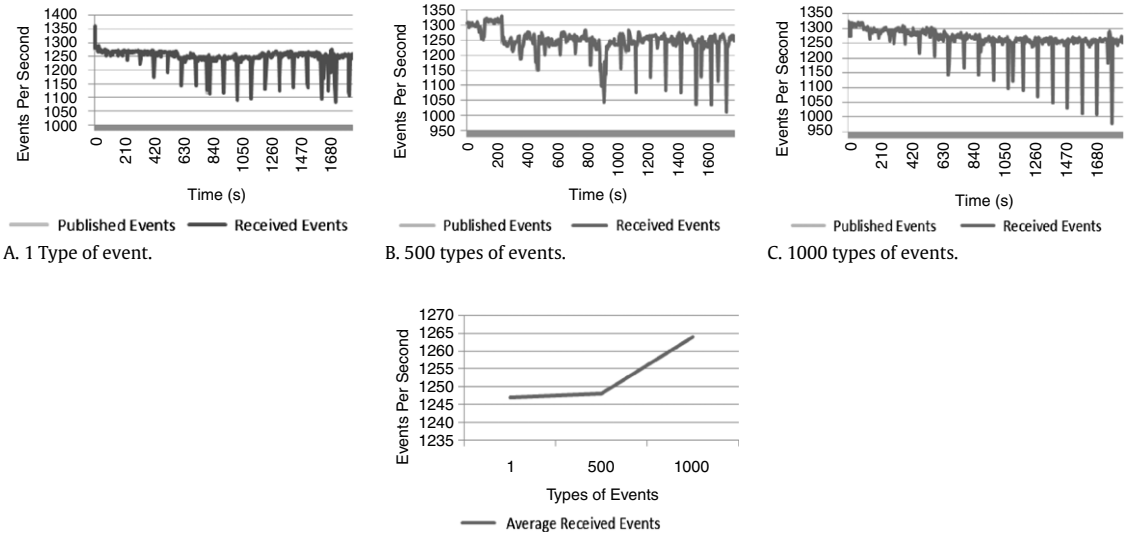

Fig. 8. Event type simulation results. Subfigures A–C show the raw throughput of the different number of event types throughout the duration of the experiment. Due to the event router building an inverted index to map event types to subscribers, the throughput is not impacted as more types of events are added. Subfigure D shows the average throughput for each number of event types for the same period.

this scenario. The performance metrics for this test were gathered from both the publishing and subscribing client nodes. Fig. 7 provides the results of the simulation. As demonstrated, when the size of the event grew 500 times, the event throughput fell by a factor of 27. Though expected, the simulation demonstrates a strong negative correlation between event size and throughput.

The next simulation examined the overall throughput of the system as the number of event types increased. We assumed this increase will have no impact on event throughput. The reasoning behind this assumption lay in the fact that the event router kept an inverted index mapping events to subscribers. In other words, while building the prototype, we assumed that it was better to make the subscription process slightly more expensive in order to allow the transmission process to be more efficient. As a result, each time an event subscription was added, an inverted index was updated. This allowed event to subscriber mapping to be done in constant time, as the number of event types increased.

The setup of this simulation consists of a single publisher and subscriber. The subscriber receives all events published. As noted above, the number of types of events published is the variable under test. The test was performed with 1, 500 and 1000 event types. Each event is a constant size of one kilobyte. All events were published on a loop containing very little logic in order to obtain maximum throughput with minimal processing latency. The performance metrics for this test were gathered from both the publishing and subscribing client nodes. Fig. 8 illustrates the results of this simulation. The results show that regardless of the number of types of events, the throughput remains constant. This is a favorable characteristic, as a typical distributed system contains an extremely diverse set of events.

The final simulation focused on altering the number of child nodes rather than the events themselves. The aim was to examine the performance of the event router as the number of active child nodes grew. Because the event router is managing more

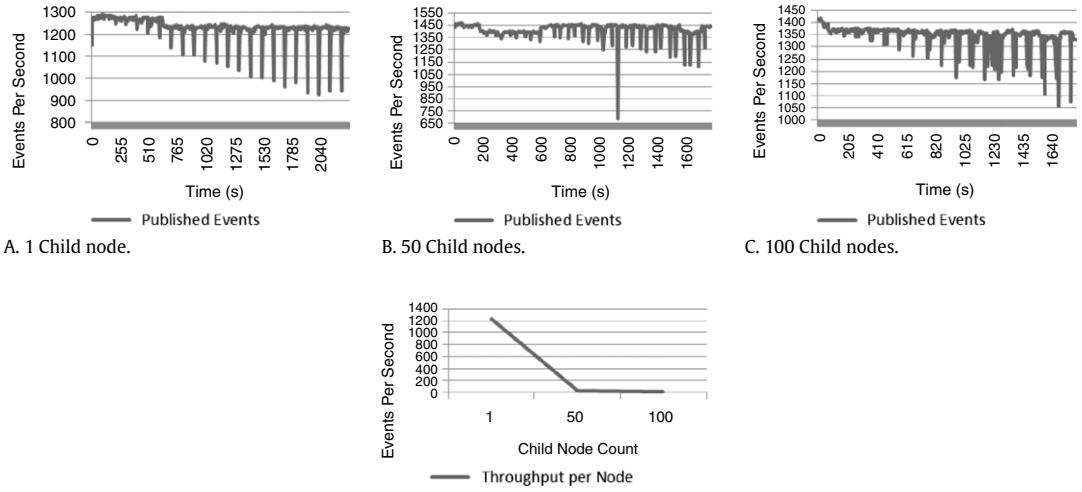

Fig. 9. Number of child nodes simulation results. Subfigures A–C show the raw throughput of an event router with differing number of children for the duration of the experiment. Subfigure D shows the average throughput per child node of the event router under test for the same period.

information as the child node count grows, it is believed that this will negatively impact event throughput. For this simulation, a single publisher was present while the number of subscribing nodes was adjusted for testing. The event size and type remained constant throughout the simulation. The test was performed with 1, 50 and 100 child nodes. The performance measurements were taken from the event router as opposed to the child nodes, as the event router serves as the bottleneck in this situation and provides a single throughput metric. Fig. 9 shows the results of this simulation. Based on this experiment, the number of child nodes did not impact event throughput at first glance. However, because the throughput was measured on the router, it was essentially saturated. This means that while the throughput remained constant, the child nodes are actually receiving fewer events as the resource consumption is divided among them.

6. Conclusions

There is a gap between the event-based middleware and CEP research. While CEP fundamentally relies on the fact that event passing capabilities be preset within a distributed system, it often exists as an independent component. Although this is still useful, the full power of CEP is not readily available to other components within the system is. The reason is because the event-based middleware being used by the system does not understand complex events, and queries cannot be formulated in such a way to take advantage of the causal relationships represented within them. Furthermore, applications cannot efficiently offload operations, such as event aggregation to the CEP engine, because they are often centrally located.

In addition to limiting the usage of CEP by the components within the system, having a disconnected event-based middleware and CEP engine make the overall system more complex. For any infrastructure component within a distributed system, there are a few vital properties that must be guaranteed in order to assure usefulness. The primary one is that of scalability. Because both the event-based middleware and CEP engine rely on different scalability models, it is more complex for a distributed system that relies upon them to scale as a whole.

Given the above issues, this article proposed an event-based middleware that treats CEP as a first class citizen. The primary hurdle to accomplishing the above is the efficiency in which events can be routed between the producers and consumers. In traditional event-based middleware systems, the functionality of the subscription language is tightly coupled to the organization of the overlay network. Additionally, only a basic set of operators are provided to allow for simpler network partitioning. However, to provide the full richness of a complex query language, the overlay network could not be designed in such a way that the supported operators would be hindered. For this reason, CERI diverges from the typically structured peer-to-peer overlay network and uses an unstructured peer-to-peer design. Furthermore, CERI places the query processing engines on the nodes themselves. Such a configuration allows components within the system to locally take advantage of the complex query language, allowing them to offload internal logic. Specifically, it is shown that as additional clients and types of events are added to the system, performance does not diminish. With these favorable properties, CERI can be scaled up to meet the needs of a growing distributed system. Our approach demonstrates that an event-based middleware solution can include CEP capabilities without making sacrifices.

References

[1] A.R. Bharambe, M. Agrawal, S. Seshan, Mercury: supporting scalable multi-attribute range queries, SIGCOMM Computer Communications Revision 34 (4) (2004) 353–366.
[2] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, Design and evaluation of a wide-area event notification service, ACM Transactions on Computer Systems 19 (3) (2001) 332–383. (2001).
[3] P. Eugster, P. Felber, R. Guerraoui, A. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys 35 (2) (2003) 114–131.
[4] A. Gupta, O. Sahin, D. Agrawal, A. Abbadi, Meghdoot: content-based publish/subscribe over P2P networks, in: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware, in: Middleware Conference, vol. 78, Springer-Verlag, New York, New York, NY, 2004, pp. 254–273. Toronto, Canada, October 18–22, 2004.
[5] B. Krishnamurthy, D.S. Rosenblum, Yeast: a general purpose event-action system, IEEE Transactions on Software Engineering 21 (1995) 845–857.
[6] J. Ledlie, J.M. Taylor, L. Serban, M. Seltzer, 2002, Self-organization in peer-to-peer systems, in: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002, EW10, ACM, New York, NY, pp. 125–132.
[7] D. Luckham, The Power of Events, Addison-Wesley, 2002.
[8] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer-Verlag New York, Inc., 2006.
[9] P.R. Pietzuch, J. Bacon, Hermes: a distributed event-based middleware architecture, in: Proceedings of the 22nd International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, 2002, pp. 611–618. July 2–5, 2002, ICDCSW.
[10] Rio de Janeiro, Brazil, June 16–20, 2003 P.R. Pietzuch, B. Shand, J. Bacon, A framework for event composition in distributed systems, in: M. Endler (Ed.), Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware, Springer-Verlag, New York, New York, NY, 2003, pp. 62–82. Middleware Conference.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker, A scalable content-addressable network, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, New York, NY, 2001, pp. 161–172. San Diego, California, United States, SIGCOMM '01.
[12] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for Internet applications, in: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, New York, NY, 2001, pp. 149–160. San Diego, California, United States, SIGCOMM '01.
[13] A. Tanenbaum, M. Van Steen, Distributed Systems: Principles and Paradigms, second ed., Pearson Prentice Hall, 2007.
[14] W.W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, A.P. Buchmann, 2003, A peer-to-peer approach to content-based publish/subscribe, in: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems, San Diego, California, June 8, 2003, DEBS '03, ACM, New York, NY, pp. 1–8.
[15] E. Wu, Y. Diao, S. Rizvi, High-performance complex event processing over streams, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, 2006, pp. 407–418. Chicago, IL, USA, June 27–29, 2006.
[16] X. Yang, Y. Hu, A DHT-based infrastructure for content-based publish/subscribe services, in: Proceedings of the Seventh IEEE International Conference on Peer-To-Peer Computing, IEEE Computer Society, Washington, DC, 2007, pp. 185–192. September 2–5, 2007, P2P.
[17] Melbourne, Australia, November 1, 2006 A.K. Yeung Cheung, H. Jacobsen, Dynamic load balancing in distributed content-based publish/subscribe, in: M. Henning, M. van Steen (Eds.), Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Springer-Verlag, New York, New York, NY, 2006, pp. 141–161. Middleware Conference.

Hossein Saiedian (Ph.D., Kansas State University, 1989) is currently an associate chair and a professor of computing and information technology at the Department of Electrical Engineering and Computer Science at the University of Kansas (KU) and a member of the KU Information and Telecommunication Technology Center (ITTC). Professor Saiedian has over 150 publications in a variety of topics in information technology, information security, software engineering and computer science. His research in the past has been supported by the NSF as well as other national and regional foundations. Professor Saiedian has been awarded a number distinguished awards, including the KU's highly prestigious Kemper award for excellence in teaching, the University of Nebraska's awards in excellence in research and excellence in teaching, and was ranked among the top 10 software engineering scholars by the *Journal of Systems and Software*. At KU, he has graduated more than 45 MS and Ph.D. students. In addition to teaching undergraduate and graduate courses, he has extensive background in industrial course development and training, and consulting. Professor Saiedian is a member of the ACM, a senior member of the IEEE, and was among the first group to become an IEEE Certified Software Development Professional.

Gabe Wishnie completed his Master's in computer science degree at the University of Kansas in 2009. His professional career began primarily focused on building large-scale web services and back office applications for various Fortune 500 companies. Seeking out new challenges, he joined the Bing organization at Microsoft in 2008. There he works within the infrastructure team to provide efficient monitoring capabilities for the distributed systems comprising Bing.