Load Balanced Telecommunication Event Consumption Using Pools

Sajeevan Achuthan
Network Management Lab, LM Ericsson,
Athlone, Co. Westmeath, Ireland
Email: sajeevan.achuthan(at)ericsson.com

Liam Fallon
Network Management Lab, LM Ericsson,
Athlone, Co. Westmeath, Ireland
Email: liam.fallon(at)ericsson.com

***Abstract*—Management systems increasingly consume events, giving them a more real-time view of the networks they are managing. Providing an event forwarding mechanism between the management system component that consumes events and application instances is a challenge. Such a mechanism must forward events in a manner that ensures correct delivery and is scalable, reliable, and efficient.**

This paper describes an approach that uses pools of event consumers and application instances to receive events being sent to a network management system from network elements in a telecommunication network. The use of pools allows the task of processing incoming events to be balanced across the members of the pool. The size of the pools can be automatically modified to cope with the current event load. The approach has been used in a proof of concept management event processing system installed on two live mobile networks.

I. INTRODUCTION

Network management systems are increasingly using events to monitor the state of telecommunication services and networks. Events sent by networks such as alarms, SNMP traps, and Charging Data Records (CDRs) have always been an important source of information for network management systems. In recent years, the number of events generated by networks has greatly increased and event-based mechanisms are now often used to provide information on network and service state and sessions [1], traditionally the domain of performance counters.

In parallel, the field of stream processing has emerged as an important technological thread in computer science. Event processing networks [2] and complex event processing (CEP) systems such as Esper [3] can be successfully applied to process event streams with high event rates. In such systems, as the event load increases, the number of event consumers is increased to scale the system to handle that increasing load.

Telecommunication events can be of many types and can come from many sources. Typical types are alarms, performance counters, session events, or CDRs. Such events may come from disparate nodes such as base stations, gateways, or routers, and may use different transport mechanisms such as 3GPP IRPs, SNMP, or NETCONF. As shown in Fig.1, a telecommunication management system receiving events typically deploys an Event Consumption Component (ECC) that can consume event streams from each incoming source and applications (A to N in Fig.1) that use events of certain

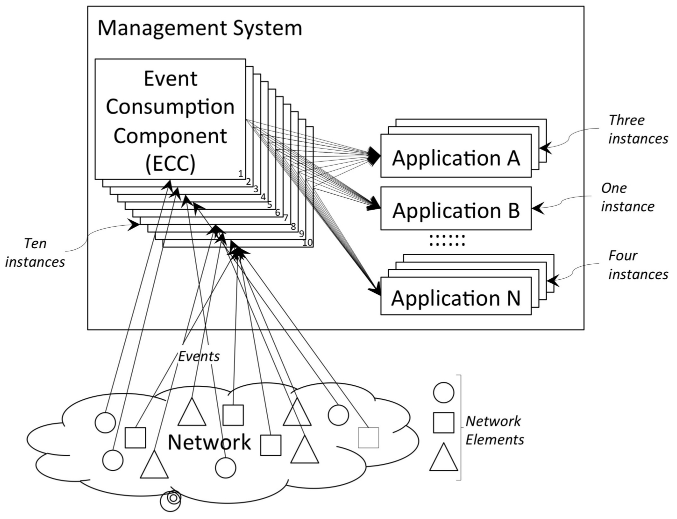


Fig. 1. Event Consumption over Consumer and Application Instances

types in order to provide some functionality. The ECC receives the events from the network elements in the network and forwards them to the applications that use them.

In order to scale a system, it is necessary to allow multiple ECC instances and multiple instances of each application to be deployed, the instances of which may be spread over a number of physical computers as shown in Fig.1. Each ECC instance handles sessions for a portion of network elements in the network, a subset of the full set of network elements in the managed network, and each application instance carries a subset of the total workload of its application.

Applications that consume events such as event correlation systems based on frameworks like Esper [3] require that specific events from a particular node must always be correlated in the same application instance. For example, multi-source events for a particular cell on a base station must all be forwarded to the same aggregation application instance so that statistics such as dropped call counts for that cell can calculated in one place.

Of course, events will always appear in the correct instance of an application if it only has a single instance, but having a single instance for each application manifestly does not scale. This paper describes a general approach that uses the concept of pooling for scalable handling of event consumption between ECC instances and application instances running in a distributed manner in a cluster. Consumer instances in ECC pools send events to application instances in application pools. The use of pools in the ECC allows the load of consuming incoming events to be balanced across the members of the pool and the use of pools in each application allows the load of each application to be spread across its running instances. The size of all pool can be modified automatically to cope with current event and application load.

The approach has been deployed in a proof of concept event processing system running in two live mobile networks. In both networks, the approach makes it unnecessary to manually configure and restart the event processing system when number of network elements and event load increases or evolves.

This paper is organised as follows. §II describes the background for this work. §III and §IV describe the approach and its implementation. §V describes our experiences in using this approach, and §VI summarises the paper.

II. Background

Systems that receive and process events are scaled by increasing the number of ECC and application instances (see §I and Fig.1). However, providing a forwarding mechanism between ECC and application instances that ensures correct delivery and is scalable, reliable, and efficient is a challenge.

ECC instances uses stateful session semantics to interact with the network elements from which it is receiving events. An instance such as an agent receiving SNMP traps [4] or an instance such as the Session Event Adapter that is receiving event streams [1] holds up long-lived sessions with a particular subset of network elements. Load balancing across ECC instances of a certain type is handled using approaches such as that described in [5], but in general load balancing is challenging because of these stateful connections.

The set of events required by a particular application such as Application B (Fig.1) is a subset of the events available in the ECC. An ECC often maintains a list of the events that should be forwarded to each application and only forwards those events to each particular application. This Forwarding List may be configured on the ECC or may be built up dynamically if the ECC provides a subscription interface for applications. Each application instance is typically assigned a subset of the problem domain to handle; a particular application instance may handle all events from a certain network element, cell, or link. Each ECC instance must send the events from a particular node to a particular application instance in each application.

In a naive implementation, every ECC instance broadcasts all events to every application instance and application instances simply ignore events in which they have no interest. Such an approach is practical in very small deployments. However, as the number of ECC and application instances increase, the event load between adapters and applications increases exponentially because of the massive duplication of events. In systems with even tens of ECC instances and application instances, the event load becomes unacceptable.

Of course, each application could implement its own forwarding mechanism, where one application instance acts as a *master* and other application instances act as *slaves*. Delegating this responsibility to applications has the disadvantage that, in order to avoid the master in each application becoming a single point of failure, each application must implement its own scalable and robust forwarding mechanism.

Another approach is to provide an interface or API that allows configuration of connections between ECC and Application instances. Manually configuring connections using a command line or GUI client towards that API is patently unscalable. Any external application developed to manage connections over such an API would have complex interactions with the management system to keep track of changes in the network element load, incoming event load, ECC and Application Instance count and modify connections between ECC and Application Instances to adjust to such changes.

Introducing an automatic mechanism in the management system for load balanced event consumption was considered the most promising approach for correct delivery of events to application instances in a scalable, reliable, and efficient manner. In order to implement such a mechanism, we considered a number of existing approaches.

Techniques such as round robin or random forwarding of events from ECC instances to application instances discussed in [6] cannot be used because each incoming event can not be sent to an arbitrary application instance. In other approaches [7] [8] [9], an application is identified from a primary key such as an IP address. The application instance in an application to which an event is sent is found by sending the event to a random application instance or by computing the primary key modulo the number of application instances. If an application instance is added or removed, events are automatically redistributed across the available application instances because of the use of the number of available application instances when computing event destinations. These approaches suffer from a number of drawbacks. They do not support multiple event consumer instances. The same event key must exist in all events being handled in an event consumer. There is no guarantee that all events from a particular source will be sent to the same application instance. Load balanced event forwarding to multiple applications that have different numbers of application instances is not possible and it is not possible to use alternative algorithms for management of connections between consumer instances and application instances.

Most messaging frameworks (eg. [10], [11]) support adaptive and configurable message consumption by clusters from queues, including dynamic and configurable load balancing. These frameworks focus on load balancing event delivery to event consumer instances and not on forwarding events to application instances; there is no inherent load balancing of events between instances in a broker cluster and application instances. In order to apply such frameworks in a way that ensures that sets of events from a given source are always sent to a particular application instance in a widely distributed application deployment, substantial adaptations would be required

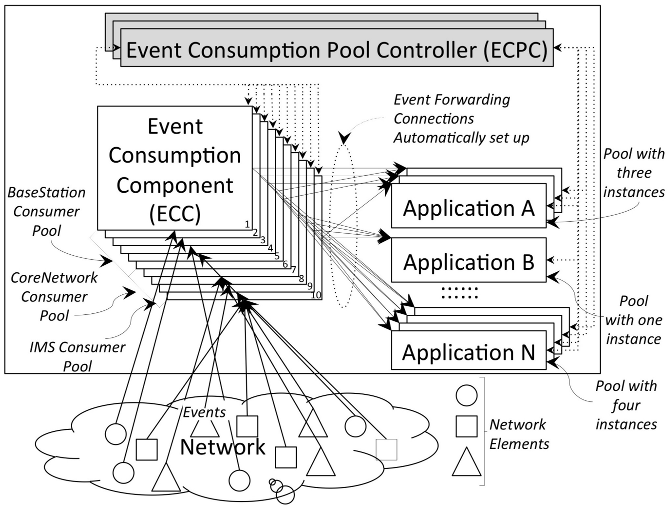

Event Consumption Pool Controller (ECPC)
Event Consumption Component (ECC)
Event Forwarding Connections Automatically set up
Pool with three instances
Application A
Application B
Application N
BaseStation Consumer Pool
CoreNetwork Consumer Pool
IMS Consumer Pool
Pool with one instance
Pool with four instances
Events
Network
Network Elements

Fig. 2. Pooled Event Consumption

to modify the frameworks. Alternatively, implementation of internal load balancing would be required in each application.

The concept of using instance pools for automatic load balancing is well known in communication networks. Instances in a pool share the incoming load, with extra instances being deployed as the load increases. An example of the use of this concept is pools of MSC nodes in 3GPP core networks [12].

III. Pooled Event Consumption

We describe a general approach for scalable handling of event consumption, shown in Fig.2, that uses pools of ECC instances and application instances running in a distributed manner in a cluster. The concept of pooling [12] is applied to the problem of scalable event consumption. Consumer instances in ECC pools send events to application instances in application pools. The use of pools in the ECC allows the load of consuming incoming events to be balanced across the members of the pool and the use of pools in each application allows the load of each application to be spread across its running instances. The size of all pools can be modified automatically to cope with current event and application load.

An Event Consumption Pool Controller (ECPC) is used to manage connections between a pool of ECC instances (consumers) and pools of application instances in each application. The ECPC automatically establishes and maintains connections between event consumers and applications so that each application receives the events it requires, the minimum number of connections is set up between consumers and applications, duplicate events are not sent from consumers to an application, and the event load is balanced over connections. Redundant standby ECPC instances may be run on separate physical hardware to ensure redundancy. Consistency can be maintained between those ECPC instances using a mechanism such as ZooKeeper [1].

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

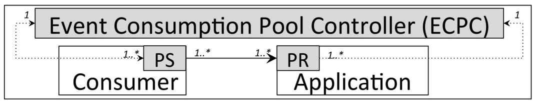


Fig. 3. Event The ECPC, Consumers and Applications

Each consumer has a type, which identifies the category of the set of events it is consuming and has available for forwarding. Consumers consuming events from a particular type of network element are of the same type. A consumer of base station events may have the type *BaseStationConsumer* and a consumer of events from core network nodes may have the type *CoreNetworkConsumer*. All consumers of a particular type are grouped into a single pool, with each consumer instance consuming events from a subset of the network elements of its type in the network. There may therefore be a *BaseStationConsumerPool* and a *CoreNetworkConsumerPool*. Approaches such as that described in [5] may be used to balance the load across instances in a consumer pool.

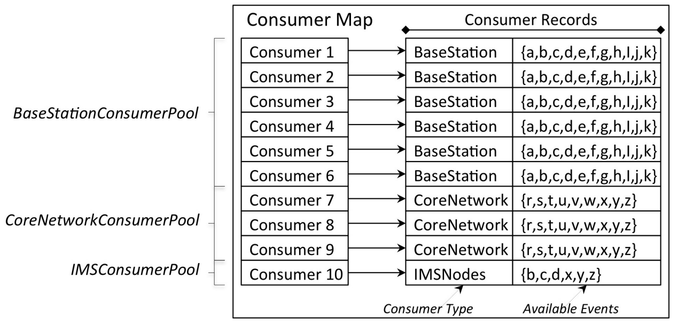
Each consumer instance has a PS (Pool Sender) that consumes and forwards events, and each application instance includes a PR (Pool Receiver) that receives events (Fig.3). The PS sends events to PRs in many different applications but each PS sends events to only one PR in a particular application. The ECPC establishes connections so that each PS in a consumer pool is connected to one and only one PR in each application to which events are being forwarded. In this way, events from all nodes being handled by the consumer pool are forwarded to each application. Events from a particular node are always forwarded to the same application instance using the PS-PR pair connection of the consumer handling that node.

When a consumer or application instance is added to or removed from a consumer or application pool, the ECPC automatically rebalances the PS-PR connections. Many algorithms for application instance to consumer instance allocation are possible. A round robin algorithm where the next consumer instance is assigned to the next application instance is described in §IV-B. A load-aware algorithm where consumer instances are assigned to application instances using actual or estimated event load measurements on consumer and application instances is explained in §IV-C. Other algorithms, such as an algorithm that considers the priority of incoming events, may also be applied.

This approach provides event-based network management systems with an event forwarding mechanism that scales flexibly. Consumer instances may be added to consumer pools and application instances may be added to application pools in a way that ensures efficient and balanced forwarding without complex manual configuration.

IV. Implementation

The Event Consumption Pool Controller (ECPC), the Pool Sender (PS) in a consumer instance, and the Pool Receiver (PR) in an application instance (see Fig.3) interact to provide balanced consumption of events. These software entities use a



Consumer Map	Consumer Type	Available Events
Consumer 1	BaseStation	{a,b,c,d,e,f,g,h,l,j,k}
Consumer 2	BaseStation	{a,b,c,d,e,f,g,h,l,j,k}
Consumer 3	BaseStation	{a,b,c,d,e,f,g,h,l,j,k}
Consumer 4	BaseStation	{a,b,c,d,e,f,g,h,l,j,k}
Consumer 5	BaseStation	{a,b,c,d,e,f,g,h,l,j,k}
Consumer 6	BaseStation	{a,b,c,d,e,f,g,h,l,j,k}
Consumer 7	CoreNetwork	{r,s,t,u,v,w,x,y,z}
Consumer 8	CoreNetwork	{r,s,t,u,v,w,x,y,z}
Consumer 9	CoreNetwork	{r,s,t,u,v,w,x,y,z}
Consumer 10	IMSNodes	{b,c,d,x,y,z}

Fig. 4. Consumer Record and Consumer Map

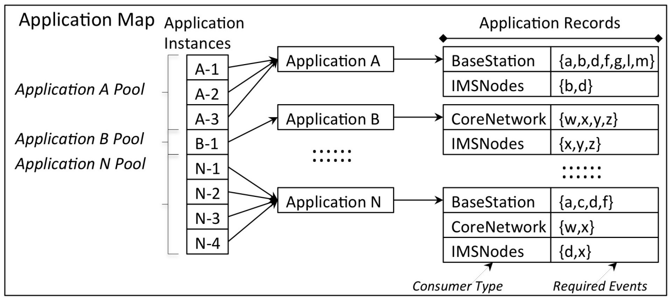
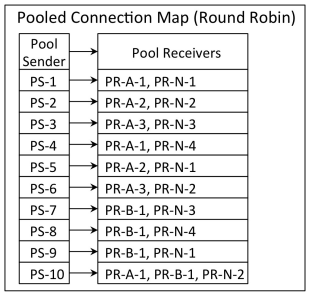


Fig. 5. Application Record and Application Map


Pooled Connection Map (Round Robin)

Pool Sender	Pool Receivers
PS-1	PR-A-1, PR-N-1
PS-2	PR-A-2, PR-N-2
PS-3	PR-A-3, PR-N-3
PS-4	PR-A-1, PR-N-4
PS-5	PR-A-2, PR-N-1
PS-6	PR-A-3, PR-N-2
PS-7	PR-B-1, PR-N-3
PS-8	PR-B-1, PR-N-4
PS-9	PR-B-1, PR-N-1
PS-10	PR-A-1, PR-B-1, PR-N-2

Fig. 6. Pooled Connection Map using a Round Robin Algorithm

synchronised inter-process communication mechanism such as Distributed Hash Tables [13] to reliably share control information, depicted as dotted lines in Fig.3. The inter-process communication mechanism distributes connection management information across the software entities and allows control messages to be passed between those entities.

An Application Instance is a member of an Application Pool. All application instances execute the same type of task and share the load of those tasks. Each application instance receives events from one or more pools of consumers. In order for an application instance task to execute correctly, it alone must receive all events from a particular set of network nodes.

All consumers in a consumer pool consume events from the same event source type. A consumer instance may forward events to more than one application; each event is forwarded to a particular application only once. A particular event instance is not sent to multiple instances of the same application.

The ECPC controls the connections between pools of consumer instances and pools of application instances. For redundancy, more than one ECPC may exist, with one ECPC nominated as master ECPC. If the master ECPC fails, a process monitoring mechanism informs the other ECPCs, and another ECPC is nominated as master. Because all connection information is reliably shared and synchronised, the new master ECPC resumes management of subscriptions from the point at which the failed master ECPC stopped management.

The data structures that are shared between the software entities are shown in Fig.4, Fig.5, and Fig.6.

Fig.4 shows the shows the Consumer Map, used to hold the type of each consumer and the list of events it has available. All consumers of a particular type are grouped together as a Consumer Pool. The *BaseStationConsumerPool*, *CoreNetworkConsumerPool*, and *IMSConsumerPool* (Fig.4) are examples. Each consumer instance is configured with its consumer type and with the list of event types it can consume. The ECPC maintains the consumer map and uses it to manage each consumer pool using the consumer type and event list sent to it by PSs of consumer instances when they start up.

The Application Map (Fig.5) holds the type and identities of events required by each application. Each application instance is configured with the consumer types and event lists it requires. The ECPC maintains the application map and creates and maintains an Application Pool for each application by reading the consumer types and event lists sent to it by each PR when they start up. Each application instance is allocated to the pool of its application.

The ECPC administers consumer instance to application instance connections using a Pooled Connection Map. Each PS sends events to a single PR in each application that requires events to be forwarded to it. The round robin allocation scheme in Fig.6 connects pool sender and receiver instances in ascending order, restarting allocation at the first instance in a pool when the last instance in the pool is reached. More complex allocation schemes, such as a scheme that considers the current load on pool instances, may also be used. The ECPC automatically manages connections between PSs and PRs by listening to notifications of changes on the consumer map and application map. When either map changes, the ECPC updates connections between consumer and application instances as appropriate using its allocation algorithm.

Fig.7 shows connections between six instances in a *BaseStationConsumerPool* and three instances in Application A pool. Here, each PR instance is fed by two PS instances. Fig.8 shows the PS of a single consumer instance in a *IMSNodeConsumerPool* feeding PR instances in Applications A, B, and N.

There are cases where a routing mechanism is required to deliver all events of a certain type are required by every application instance, for example all instances of application A may require Service Class events from IMS nodes in addition to Base Station events. This routing mechanism is outside the scope of this paper and is the subject of future work.

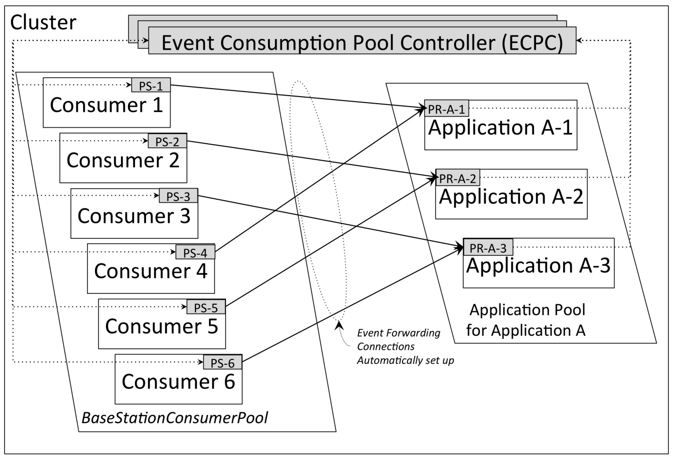


Fig. 7. Pooled Connections of Application A

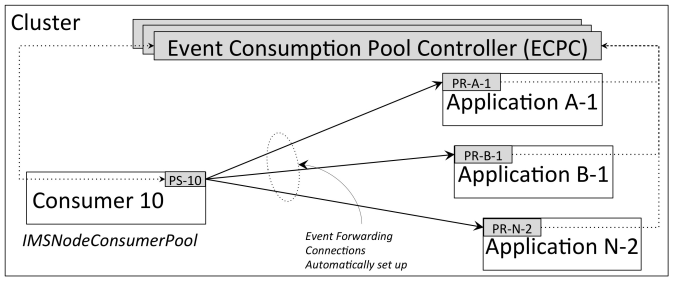


Fig. 8. Pooled Connections of Consumer 10

A. *ECPC Handling of Consumer or Application Map Changes*

If an application instance is added or removed, the ECPC builds a list of all consumer pools that are feeding the application pool in which the change occurred. It iterates over the list of consumer pools and reallocates connections between each consumer pool on the list and the application pool.

If a consumer is added to or removed, the ECPC builds a list of all application pools that are being fed by the consumer pool in which the change occurred. It iterates over the list of application pools and reallocates the connections between each application pool on the list and the consumer pool.

The algorithms in §IV-B and §IV-C are two algorithms for connection allocation. Other algorithms could use real measurements of load or estimations of load on consumer or application instances or differing capabilities of the hardware on which instances are running (such as differing CPU, memory bandwidth), or even spatial information to spread the event load more fairly.

B. *Consumer to Application Pool Allocation: Round Robin*

Algorithm 1 performs a straightforward round-robin allocation. All connections between the consumer pool and application pool are first cleared. The algorithm iterates across the consumer pool instance list and creates a connection between the next PS in the consumer pool and the next PR in the application pool as they appear in each list. If the end of the application pool list is reached, the algorithm selects the first instance as the next instance on the application list.

Algorithm 1 Round Robin Connection Allocation

```
clear existing Consumer to Application Connections;
while not at end of Consumer Pool do
    $cpi \leftarrow$ next Consumer Pool Individual;
    $api \leftarrow$ next Application Pool Individual;
    if end of Application Pool Reached then
        $api \leftarrow$ First Application Pool Individual;
    end if
    connect $cpi$ to $api$;
    record connection in Pooled Connection Map;
end while
```

This algorithm ensures that each consumer instance in a pool is connected to one and only one application instance in each application that requires events from that consumer pool. There is no requirement for all application instances in an application pool be connected to a consumer instance in a consumer pool as long as all consumer instances are connected to one and only one application instance.

In order to connect a PS to a PR, an application instance is informed of the address of its consumer instance and a consumer instance is informed of the address of its application instance. The connection is recorded in the connection map.

C. *Consumer to Application Pool Allocation: Load Aware*

Algorithm 2 Load Aware Connection Allocation

```
if new Application Instances then
    move most loaded connections to
    new Application Instances;
else if deleted Application Instances then
    move deleted Application Instance connections to
    least loaded remaining Application Instances;
end if
if new Consumer Instances then
    Create connections from new Connection Instances
    to least loaded Application Instances;
else if deleted Consumer Instances then
    move most loaded remaining connections to Application
    Instances that handled deleted Connection Instances;
end if
```

Algorithm 2 implements load aware allocation of connections between PSs in consumer pool instances and PRs in application pool instances.

When application instances are added to a pool, the algorithm examines the PS-PR connections of all instances in the application to determine which connections are carrying the heaviest load. A configurable percentage of these connections are moved to the new application instances. Each reallocated PS-PR connection is disconnected from its current application instance and re-connected to a new application instance. When application instances are removed, the algorithm moves and distributes the PS-PR connections of the application instances being deleted across the least loaded application instances.

When new consumer instances appear, the algorithm creates PS-PR connections from each new consumer instance to the least loaded application instances. When consumer instances are deleted, the algorithm moves a configurable percentage of the most heavily loaded connections to the application instances that were carrying load for deleted consumer instances.

This algorithm, although more complex than the round robin algorithm described in §IV-B, has the advantage of causing much less disruption during connection changes.

D. Event Consumption

When an event is received by a consumer instance, it is immediately sent to each Pool Receiver connected to the Pool Sender of that consumer instance if the event is on the list of events requested by the application instance to which the pool receiver belongs. Forwarding is straightforward and efficient because all connections between a PS and the PRs to which it is forwarding have already been established by the ECPC. Because of its simplicity, the execution time and resource usage for event consumption in the approach is very low.

V. Deployment and Experiences

This approach is deployed in a proof of concept event processing system in operator networks in the Americas and East Asia.

In the American deployment, the operator's network, which has tens of thousands of nodes, is growing at the rate of 5% per week. Prior to deployment of this approach, static configuration of connections between ECC and Application Instances and a system restart was required to add network elements. As a result, network elements were added to the system weekly at night time in the maintenance window. Since deployment of this approach, network elements are added to the system as they are installed and are handled immediately by the system without a system restart. The time taken for the system to adjust to the appearance of a new network element is of the order of 1 second and the system has not been restarted since the approach was deployed three months ago.

In the East Asia deployment, the network is also growing rapidly in size and the event load is rising; the number of subscribers is increasing at a rate of 20% per month. Manual configuration and activities in the maintenance window are no longer necessary to introduce new ECC and Application Instances to handle this extra load, they are now deployed on demand and the system adjusts elastically to this extra load. In this deployment, as in the American deployment, system adjustments take of the order of 1 second and system restarts are no longer necessary for configuration changes.

VI. Summary

The approach described in this paper for load balanced event consumption using pools automatically handles connections between pools of event consumers and pools of application instances in a scalable manner. It allows consumer pools and application pools to be independently sized, allowing event-based network management systems to scale easily.

No manual configuration of event forwarding is required, pooled connections are managed automatically. The approach scales well because events are not duplicated and are only forwarded from consumers to applications that require them; unused events are not forwarded. The approach can automatically adapt to changes in load experienced by consumer and application instances by automatically adapting pool sizes. Consumers in a consumer pool, application instances in an application pool and ECPCs are redundant. If a consumer, application instance, or ECPC fails, the other running consumer, application instances, or ECPCs automatically absorb the failed entity's load. Applications need not create their own load balancing forwarding mechanisms, this approach is a common forwarding mechanism available for all applications. Event forwarding is straightforward and fast because all connections between Pool Senders and Pool Receivers are pre-configured; no routing based on the content of events is required.

The approach has the drawback of being more complex to implement than a manual or an event broadcasting approach. In addition the deployment and management of ECPC instances must be managed. However, our experience has been that this increase in implementation complexity is offset by the reduction in complexity in run-time management of event forwarding that the approach brings.

In future work, we will investigate how specific events required by some or all application instances may be forwarded to those application instances.

References

[1] P. Gustas, P. Magnusson, J. Oom, and N. Storm, "Real-time performance monitoring and optimization of cellular systems," *Ericsson Review*, no. 1, pp. 4–13, January 2002.

[2] O. Etzion and P. Niblett, *Event Processing in Action*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[3] EsperTech, *Esper Reference*, 4th ed., EsperTech, Sep 2012. [Online]. Available: http://esper.codehaus.org/esper/documentation/documentation.html

[4] IETF, "An architecture for describing simple network management protocol (snmp) management frameworks," RFC 3411, IETF, Tech. Rep. RFC 3411, Dec. 2002.

[5] L. Fallon and S. Achuthan, "An Algorithm for Load Balancing in Network Management Applications Managing Virtualised Network Functions," in *Network and Service Management (CNSM), 2014 10th International Conference on*, Oct 2014, p. To Appear.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[7] LinkedIn Databus. (2014) Databus load balancing client. [Online]. Available: https://github.com/linkedin/databus/wiki/Databus-Load-Balancing-Client

[8] Apache Kafka. (2014) A high-throughput distributed messaging system. [Online]. Available: https://kafka.apache.org/documentation.html

[9] jboss HornetQ. (2014) User manual. [Online]. Available: http://docs.jboss.org/hornetq/2.2.14.Final/user-manual/en/html_single/

[10] Apache ActiveMQ. (2014) Messaging and integration patterns server. [Online]. Available: http://activemq.apache.org

[11] Pivotal RabbitMQ. (2014) Robust messaging for applications. [Online]. Available: https://www.rabbitmq.com/

[12] "Intra-domain connection of Radio Access Network (RAN) nodes to multiple Core Network (CN) nodes," TS 23.236, 3GPP, Jun 2013.

[13] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, 2012.