A Paradigm Comparison for Collecting TV Channel Statistics from High-volume Channel Zap Events

Pål Evensen*
paal.evensen@altibox.no
Hein Meling
hein.meling@uis.no
Department of Electrical Engineering and Computer Science
University of Stavanger, Norway

ABSTRACT

The current approach used to obtain official television channel statistics is based on polls combined with specialized reporting hardware. These are deployed only on a small scale and batch processed every 24 hours. With the enhanced capabilities of present-day IPTV set-top-boxes, network operators can track channel popularity and usage patterns with a degree of precision and sophistication not possible with existing methods. One such network operator, Altibox, is the largest provider of IPTV in Norway with a deployment of over 320,000 set-top-boxes. By tapping into the high-volume stream of channel zap events sent from these set-top-boxes, very accurate viewership can be obtained and presented in near real-time.

In this paper, we examine two programming paradigms for implementing applications to compute viewership based on channel zap events. One based on a general-purpose programming language (Java) and the other based on a highly specialized event stream processing language (EPL). An important characteristic of this application is stateful event processing. We are interested in exploring the trade-offs between these two implementations, to determine their suitability for such applications. Specifically, we are interested in the performance trade-off and the program complexity of each implementation.

Our results show that a pure Java implementation has a significant edge over EPL in terms of performance. Although, our numbers cannot be used to draw a general conclusion, it seems indicative that an event stream processing engine would suffer more than a general-purpose language as query complexity grows. We conjecture that this is because it is easier to construct custom data structures for the specific problem in a general-purpose language like Java. In terms of program complexity, EPL has a slight edge in all metrics, and a significant edge when event streams can be reused to perform more complex processing, indicating that less effort is necessary to extend functionality.

*Pål Evensen is also with Altibox.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'11, July 11–15, 2011, New York, New York, USA.
Copyright 2011 ACM 978-1-4503-0423-8/11/07 ...$10.00.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*distributed systems*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Experimentation, Measurement, Performance

Keywords

TV viewership statistics, Stream processing

1. INTRODUCTION

Viewer statistics is the most important metric used by television broadcasters to plan their programming, and for many broadcasters, to rate their advertisement time slots. Gaining an improved understanding of viewer behavior and responses to the current programming is essential to a successful TV channel. The state-of-the-art approach to obtain the viewership of a program is to *sample* a very small *selected, but hopefully representative* portion of the population. In Norway, the sample size is 1,000 out of 2,000,000 television households (0.05 %) [11], while in the US, only 25,000 out of 114,500,000 households are sampled (0.02183 %) [18]. Such a small sample size is often criticized as being statistically insignificant [18], and may lead to incorrect conclusions about actual viewer interests in a specific program, and viewer exposure to advertisements.

With the enhanced capabilities of present-day IPTV set-top-boxes (STBs), network operators can track channel/program popularity and usage patterns with a degree of precision and sophistication not possible with existing methods. This can be done by recording or aggregating channel change events (also called *zap events*) from customer STBs. Hence, assuming that the network operator's customers represents a statistically significant portion of the population, collecting statistics based on zap events is likely to provide a much more accurate statistic compared to state-of-the-art.

There are generally two approaches to compute accurate viewership. One is to store every zap event for later bulk processing, e.g. using transactional databases or techniques based on Map-Reduce [8, 15], or aggregate statistics can be computed on-the-fly, based on in-memory state. We take the latter approach, as we are mainly interested in aggregate information from these events and want to avoid storing huge volumes of data. Only aggregate numbers are stored on disk or forwarded to interested parties.

In this paper, we describe the architecture in which STBs are deployed, and how channel zap events are propagated to an aggregation cluster for online incremental event processing. Based on this architecture, our goal is to analyze the capabilities and trade-offs between two programming paradigms for building our application to obtain viewership statistics. Hence, we have implemented two applications that compute two different statistics based on received zap events: (i) the number of viewers for all channels, and (ii) detecting 15 % rise/drop in the viewership for a channel. The first is used to generate a *top-ten* list of the most popular channels/programs in near real-time. The second application reveals useful information about which programs are luring people away from other channels. An important characteristic of both these applications is that they are stateful and demand significant computational resources to ensure timely processing. Although the applications that we cover here are fairly simple, we have already implemented several other interesting incremental statistical measures that network operators and broadcasters might find interesting, and we expect to publish those results in follow-up work.

The two applications have been implemented in two very different programming paradigms. One based on the general-purpose object-oriented programming language Java, and the other based on the Event Processing Language (EPL) [9, 17]. EPL is highly specialized declarative event stream processing language derived from SQL. We are interested in exploring the trade-offs between these two paradigms, to determine their suitability for our applications. Specifically, we are interested in the performance trade-off and the program complexity of each implementation.

Java is expected to have higher program complexity than EPL, since EPL is specifically designed for processing events. We compare our implementations using on several metrics for analyzing code complexity, including lines of code and Halstead's complexity measure [13, 6, 12], in addition to a more subjective discussion based on our experience developing these applications. The results indicate that EPL might yield easier reuse compared to Java.

Previous, but simple, benchmarks using EPL [19, 10], and running the EPL benchmark kit, have indicated that it would offer competitive performance. However, at the outset of this work it was not clear if EPL would offer competitive performance for our somewhat involved applications. Our performance evaluation involves data obtained from more than 250,000 STBs. We conduct both memory profiling and throughput analysis, and find that our implementations of these applications have very different performance characteristics in the two programming paradigms.

Paper organization: Section 2 introduce the problem of obtaining viewership statistics and surveys the state-of-the-art techniques used. In Section 3 we describe the architecture of our current deployment, and outline plans for improving the accuracy of statistics, and provide new services to customers. Focusing on the event processing logic, Section 4 describe the details of our applications, and their implementation in the two programming paradigms. In Section 5 we give a brief analysis of the viewer statistics obtained from our current deployment. Additionally, we evaluate our implementations in terms of throughput and memory usage, and program complexity. Section 6 surveys related work. Finally, Section 7 concludes the paper with an overall discussion of the merits of the two paradigms.

2. BACKGROUND

We begin by describing the current state-of-the-art in measuring viewership and program rating, focusing on the Norwegian television market. This is followed by an architectural overview of the network infrastructure used by an IPTV network operator in Norway.

TNS Gallup [23] is a Norwegian company that specializes in polls and ratings, and is the main provider of viewership to the official television networks in Norway. To measure the viewership, a device dubbed the *mediameter* is used to record and log inaudible sonic signatures emitted from the audio part of television and radio programs that the device is exposed to. The participants in this continuous poll are required to *carry the device with them*, and to keep the sound audible in order for the *mediameter* to record appropriately. The device must then be placed in a docking station overnight, to transmit the recorded data to TNS Gallup. In addition to *mediameter*, TNS Gallup also collect data from 1,000 selected households whom have a specialized logging device attached to their television, but still requires operating a special remote to record changes. The device records viewer data and transmits these every night. Viewership is computed from the collected data, where each household supposedly represents 2,000 households from the same district. This type of continuous polling represents the state-of-the-art in obtaining viewership, and similar systems are deployed in many other countries, including the US [18]. Anecdotally, non-technological approaches like *viewer diaries* are apparently also still being used [18].

Altibox [4] is the largest distributor of television over a pure IP-based network in Norway, with a deployment of over 320,000 STBs, distributed amongst approximately 300,000 households. Customers are connected to two main distribution centers by fiber-to-the-home, giving customers a unique bandwidth capacity to support a variety of services, including Internet, Voice over IP telephony, IPTV, Video-on-Demand (VoD), and Personal Video Recording (PVR). The STB is the host device for IPTV, VoD, and PVR services, and to simplify interacting with these services, an Electronic Program Guide (EPG) is also available to users. Technically, the EPG is essentially a database accessible through a web service interface that associate channel name to information about the programming of that channel. Currently, Altibox offers a total of 253 TV channels, accessible through the STB by way of IP multicast (through their fiber-based broadband network). The software on STB devices are regularly updated with new service offerings, bug fixes, and QoE monitoring and diagnostics applications [14, 1]. Moreover, since STBs also have two-way communication capabilities, network operators can update their functionality to track program popularity and usage patterns by recording zap events. This can take place without any changes to current user behavior, such as using a special remote or carrying a *mediameter*. Thus, data collection is transparent to the user, and the reported data is expected to be more accurate, since users cannot forget to record the change. Moreover, we also avoid the embarrassment factor sometimes present in surveys, where users report an idealized version of their habits due to embarrassment over their factual habits. It is not unthinkable that this factor can play a role when viewers decide whether to use the special remote when viewing programming that is perceived as of lesser quality. Finally, it also allows for a much more accurate understanding of

viewer behavior than existing methods, as the sample size is more than 300 times larger, representing approximately 16 % of the Norwegian television population.

2.1 Event Processing Language

Here we briefly survey the capabilities of the EPL language and its runtime environment, referred to as the Esper processing engine. Esper [10] provides an open-source implementation of the EPL processing engine, and the necessary Java libraries for interacting with it. Our choice of Esper and EPL is primarily motivated by its focus on stream processing, and secondary its open-source licensing model.

EPL is a declarative query language derived from SQL; it shares much of its syntax and functionality with SQL, such as select, insert, update, and aggregation functions for summation, averaging, and join operators. However, instead of operating on relational database tables, EPL operates on streams of data. Using these operators, one can construct a wide variety online queries that can be used process data from event streams, such as the stream of channel zaps from customer STBs. An EPL query will process one or more event streams, looking for event patterns that match the query, and produce an output event. Moreover, since streams are continuous, i.e. not temporally restricted, EPL introduces a sliding window concept to be able to construct queries that operate over limited, but sliding, time intervals. This is used in our implementation of rise/drop detection.

Esper can handle events represented in a variety of ways, e.g., as Java or C# objects that provide getter and setter methods to access its attributes, and is the approach used in this paper.

Deploying an Esper server typically involves the following steps: (i) start the Esper processing engine, (ii) install EPL queries, (iii) establish subscriptions by registering listener objects with the Esper processing engine, and (iv) receive and parse events from the data stream, and construct Java objects to be passed to the processing engine to be processed by the installed queries. The subscriptions are connected with the installed queries, acting as handlers for output events generated by the queries.

3. ARCHITECTURE

In this section, we present the architecture of our current deployment, and discuss some changes that we are planning to implement in the near future. These changes will significantly improve the accuracy of our measurements, at the expense of more demanding processing and network overhead. We also outline a few applications that become possible with more accurate measurements.

3.1 Current Deployment

The STB devices deployed in customer residences for supporting IPTV, VoD, and PVR, are fully capable of two-way communication, and have been augmented with a software agent to keep track of and report zap events to a centralized server. We call these the ZapReporter client and ZapCollector server, respectively. A simplified architecture is illustrated in Figure 1.

The ZapReporter monitors channel changes performed by the user of the STB, and generates zap events containing the following information:

⟨Date, Time, STB-IP, ToChannel, FromChannel⟩

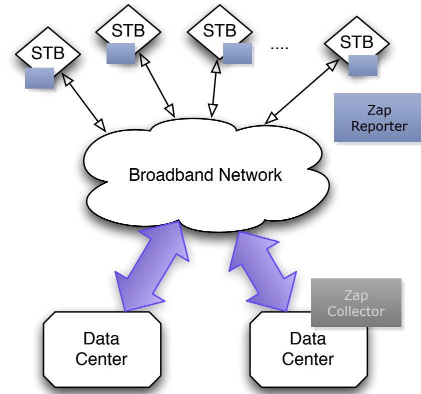


Figure 1: Network architecture illustrating STBs.

The event is encoded as text, and one event is typically less than 60 bytes, hence approximately 25 events can be sent in one 1500 byte message. The clocks on the STBs are synchronized using NTP, and thus provide more accuracy than what is needed for our purposes.

Events are generated according to Algorithm 1, and described informally as follows. When a user change channel, the ZapReporter record this change event locally on the STB and starts a timer. If the user stay on the same channel longer than 60 seconds, the event is saved away in *unsent*. If the user change channel again before the 60 second timer expires, the event is overwritten (i.e. not recorded in *unsent*). Periodically, the events stored in *unsent* is sent off to the ZapCollector and emptied.

Algorithm 1 ZapReporter pseudo code

1: **Initialization:**
2: $T \leftarrow 60$ {Timeout period (seconds)}
3: $S \leftarrow 1$ {Period of between sends (hours)}
4: $event \leftarrow \perp$ {Most recent event, not yet recorded in *unsent*}
5: $unsent \leftarrow \emptyset$ {Set of unsent zap events}
6: startPeriodicTimer(⟨SendTimeout⟩, S)

7: **on** ⟨ChannelChange, $toCh$, $fromCh$⟩
8: $event \leftarrow$ preparEvent($toCh$, $fromCh$) {Update event}
9: restartTimer(⟨RecordTimeout⟩, T)

10: **on** ⟨RecordTimeout⟩
11: $unsent \leftarrow unsent \cup event$ {Record event}

12: **on** ⟨SendTimeout⟩
13: $\forall e \in unsent$: **send** ⟨ZapEvent, e⟩ to ZapCollector
14: $unsent \leftarrow \emptyset$

This strategy ensure that the total number of messages sent are kept to approximately one message per hour per STB, and at most 60 events needs to be kept in STB memory. We expect that we would rarely see more than 25 events generated by the same STB in one hour, requiring more than one 1500 byte message to be sent. Hence, in the worst case, when all 320,000 STBs are active, we might see a total of 320,000 messages per hour, or just over 1 Mbps (on average). Moreover, if all messages contain 25 zap events, the processing rate would have to be about 2.2k events/second.

On the server side, the ZapCollector collect events

from all the STBs and store them in log files that are rotated daily. The events are stored in the order they are received from the STBs. However, since each message contains about one hour worth of zap events, the log files are not initially sorted by the timestamp. Therefore, the event logs must be sorted before they can be used to produce incremental statistics. Currently, there are no service offerings at Altibox that take advantage of these log files, but next we explain our plans for extending this to provide more accurate statistics and near real-time updates of these statistics.

3.2 Planned Deployment

There are several reasons why we are interested in increasing the accuracy of these statistics. First of all, we want to be able to provide a ranking (top-10 list) of programs in near real-time to both viewers and broadcasters. Also, we are interested in detecting flash crowds, i.e. when a large number of viewers change to or from the same channel within a short period of time. This might be expected either when a new (popular) program is beginning, or during commercial breaks. The former we have seen evidence of from our current datasets. However, to understand better the user behavior in commercial breaks, we need more accurate information from the ZapReporter. Also to provide a real-time ranking, we must to revise the ZapReporter.

Thus, in the planned deployment we are aiming to report channel changes (lasting 3 seconds or more) within a 10 second interval. Thus, in Algorithm 1, we set $S = 10$ seconds, and $T = 3$ seconds. Obviously, no message will be sent if there are no channel changes. To determine the worst case network resources necessary with this sampling frequency, assume all 320,000 STBs generate 3 events every 10 seconds. Assuming every event takes 60 bytes, the packet size should be roughly 180+70 bytes (including headers). Under these assumptions, the worst case network load would be 64 Mbps overall, and 96,000 events/second would have to be processed. These numbers are obviously above what is expected in the normal case, but we would like to be able handle flash crowds that might reach towards such numbers.

Note that, the ZapReporter functionality implemented in STBs is beyond the direct control of the authors of this work. However, we can influence and request implementation changes to the STBs. The reasons for this is corporate policies relating to accountability for changes that can potentially cause problems for customers. Moreover, the STB can only be updated two times a year, during a relatively short time window. Hence, this poses some challenges for us in implementing the desired functionality.

On the ZapCollector end, we instead introduce a ZapProcessor to process events incrementally to compute statistics for program ranking in near real-time, and for detecting flash crowds and other similar statistics. We have implemented these services and in Section 5 we evaluate our ZapProcessor implementations in both Java and EPL, based on real data obtained from our log files.

4. EVENT PROCESSING

In this section we present our two applications for obtaining viewership statistics and detecting sudden changes of viewership on a channel. We describe their implementation in both Java and EPL, specifically focusing on the event processing aspect.

4.1 Viewer Statistics in Java

Algorithm 2 gives an overview of the ZapProcessor implementation to obtain viewer statistics. Lines 10-13 of Algorithm 2 checks to see if the STB have been active in the past, and if so replaces the $fromCh$ field of the message with the last recorded previous channel. This is necessary because not all channel changes are propagated to the ZapProcessor, due to the 1 minute rule or even the 3 second rule imposed by the ZapReporter. Otherwise, our counting in the last part would not be correct. In order to obtain statistics for the different channels, we simply count the occurrences of zap events changing to the different channels. We implement this using a multiset, where each entry (the channel) is associated with a count value representing the number viewers on that channel. Moreover, we also have to reduce the count for the channel the STB is moving away from (or the previously recorded channel of that STB). We do not reduce the count of any channel if the event originate at an STB from which we have no recorded events.

Algorithm 2 ZapProcessor pseudo code

```
 1: Initialization:
 2: R                                   {Subscribers of output events}
 3: EPG                           {Electronic Program Guide database}
 4: S ← 10                   {Period of between output events (seconds)}
 5: STBs ← ∅                          {Set of known STB-IP addresses}
 6: viewers ← ∅            {Multiset: viewer count for each channel}
 7: prevCh ← ∅               {Map from STB-IP to previous channel}
 8: startPeriodicTimer(⟨OutputTimeout⟩, S)
 9: on ⟨ZapEvent, date, time, ip, toCh, fromCh⟩
10:    prev ← prevCh.get(ip)        {Get previous channel of ip}
11:    if prev ≠ null then
12:       fromCh ← prev
13:    prevCh.put(ip, toCh)  {Update previous channel of ip}
14:    viewers.add(toCh)              {Increase count of toCh}
15:    if ip ∈ STBs then             {Have we seen STB before?}
16:       viewers.remove(fromCh)        {Reduce count of ch.}
17:    else
18:       STBs.add(ip)                   {New STB, record ip}
19: on ⟨OutputTimeout⟩
20:    topCh ← viewers.mostFrequent(10)          {Top-10 ch.}
21:    for ch ∈ topCh
22:       prog ← EPG.getProgram(ch)               {Query EPG}
23:       topProgList.add(prog)  {Create top-10 program list}
24:    send ⟨Top10List, topProgList⟩ to R
```

Periodically, output events are generated by first determining which channels have the most viewers, and for each channel query the EPG to determine which program is currently being broadcast on that channel. To avoid frequent database queries, we cache program information in memory. From this we construct the top-10 list of programs to be sent to interested subscribers, providing near real-time viewership information. One such subscriber that we have implemented is the EPG itself. In this case, we integrate the top-10 list within the program guide interface on the STB device, enabling users to viewer statistics and choose program from the list.

An important improvement that these real-time viewer statistics provide over batched statistics is that broadcasters could potentially adjust their advertisement programming

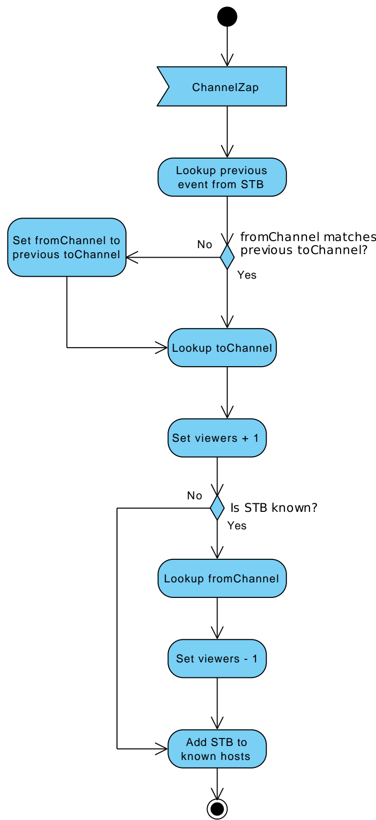


Figure 2: Viewer Statistics Activity Diagram

based on actual viewer numbers, as opposed to predicted number of viewers.

4.2 Viewer Statistics in EPL

Both implementations share a common overall logic in how events are handled (see Figure 2 and Algorithm 2). However, the EPL implementation requires a slightly different understanding of how events are related, and hence the following gives a more succinct description from the EPL perspective, while the algorithmic descriptions closely match the Java implementation.

As shown in Figure 2, incoming events are matched against the previous event received from the same STB, comparing its *fromChannel* field with the *toChannel* field of the STB's previous event. Different values might be observed at this stage, either due to packet loss, or more likely due to the way that ZapReporter generate events (not all events are actually sent). To compensate for this, we set the *fromChannel* of the incoming event to the *toChannel* of the previous event. If no previous event exist, no action is taken.

The next step is updating the number of viewers by adding one to the channel matching the *toChannel* field and subtracting one from the channel matching *fromChannel*. If the event received is from a previously unknown STB, the *fromChannel* is not subtracted, as it is the first event received from this particular STB. Finally, the STB is added to the list of known devices.

The EPL queries contains all of the logic depicted in Figure 2, while the EPL implementation also requires some Java code to handle parsing and object creation for incoming events. Also, listener objects must implement a callback interface in Java to receive output events generated by the Esper engine. We have not included the Java code.

Listing 1 EPL Viewer Statistics

```
create schema ChannelTotViewers
  as (channelName string, viewers int
create window ChannelWin.std:unique(channelName)
  as ChannelTotViewers
create window StbWin.std:firstunique(ip)
  as tv.ChannelZap
create window ZapWin.std:unique(ip)
  as tv.ChannelZap

insert into ZapWin
  select * from tv.ChannelZap

update istream tv.ChannelZap as zap
  set fromChannel =
    (select toChannel from ZapWin where ip = zap.ip)
  where fromChannel !=
    (select toChannel from ZapWin where ip = zap.ip)

on tv.ChannelZap zap merge ChannelWin cw
  where zap.toChannel = cw.channelName
  when matched
    then update set viewers = viewers + 1
  when not matched
    then insert
      select toChannel as channelName,
      1 as viewers

on tv.ChannelZap zap merge ChannelWin cw
  where zap.fromChannel = cw.channelName and
    exists (select * from StbWin where ip = zap.ip)
  when matched
    then update set viewers = viewers - 1

insert into StbWin select * from tv.ChannelZap

insert into ZapSnap
select *, percent(viewers, sum(viewers)) as activity
  from ChannelWin
  output snapshot every 15 sec
  order by viewers desc
```

Listing 1 shows the complete EPL code for generating viewer statistics. The `tv.ChannelZap` variable refers to the Java object created when parsing incoming events. The **`update istream`** query is necessary to compensate for any discrepancy in from/to channel values, as described above, and operates on the `tv.ChannelZap` event before it enters any stream. In addition, the code updates a `ChannelWin`, containing viewer numbers, as well as adding the STB to the `StbWin`, containing known STBs. The reason for using the whole `tv.ChannelZap` object most of the time, instead of extracting only the necessary values is that according to the Esper documentation [9], selecting individual properties from an underlying event object comes with a performance penalty, as the engine must then generate a new output event containing exactly the selected properties. Additionally, it simplifies the syntax. The final statement in Listing 1 outputs an ordered snapshot of the channel window every 15 seconds, decorating it with a percentage value, calculated by a custom method implemented in Java.

We ran both the Java and EPL implementations over the same datasets, and after a few rounds of debugging, we observed identical output for both implementations.

4.3 Annoyance Detection in Java

Next, we discuss our last application which is aimed at detecting if a particular ad is causing viewers to change channel. Broadcasters would most likely want to know about this, in order to remove or charge more for ads that annoy or upset viewers. To support such ad annoyance detection, we must detect changes in the viewership beyond some threshold, e.g. measured as a fraction, P, of the total number of viewers on that channel. Algorithm 3 shows the additional code necessary for such detection. To implement this, we again rely on a multiset to keep a count of the number of zap events seen in the current interval. The interval used in this case is 60 seconds, but this can easily be adjusted for more fine grained intervals. Note that *ival* is an integer, and the + symbol represents concatenation. Hence, the element of the multiset is the concatenation of channel name and an integer representing an interval. To ensure that memory usage is kept low, we immediately expunge data from a previous interval, and if an output event is generated within one interval, we reset the counting for that interval. This allows multiple output events to be generated for the same interval, if the fraction of viewers changing channel in that interval is $\geq 2P$.

Algorithm 3 Annoyance detection pseudo code

1: **Initialization:**
2: F {Multiset: count viewers moving from ch. in intervals}
3: $M \leftarrow 2000$ {Minimal # of viewers to consider for detection}
4: $P \leftarrow 0.15$ {Fraction of viewers moving from ch. in interval}
5: $prevIval \leftarrow \perp$ {The previous interval}

6: **on** $\langle$ZapEvent$, date, time, ip, toCh, fromCh\rangle$
7: $ival \leftarrow time/60$ {Get interval of this event (sec)}
8: **if** $ival \neq prevIval$ **then**
9: F.clear() {New interval begun; expunge old entries}
10: $prevIval \leftarrow ival$
11: F.add($fromCh$+$ival$) {Inc. count changing from ch. in ival}
12: F.remove($toCh$+$ival$) {Reduce count for ch. moved to in ival}
13: $v \leftarrow viewers$.count($fromCh$) {#Viewers on fromCh}
14: **if** $v > M \wedge F$.count($fromCh$+$ival$)$\leq P \cdot v$ **then**
15: Generate output
16: F.setCount($fromCh$+$ival$, 0) {Reset count for ival}

4.4 Annoyance Detection in EPL

The annoyance detector in Listing 2 looks at the average viewer number over the last minute, constantly comparing the most recent number with the average. If the viewer number drops with 15 % compared with the last minute average, an output event is triggered.

Here, the power of sliding time windows are illustrated: It selects some properties from a sliding time window, operating on the `ZapSnap` stream of viewer statistics. `ZapSnap` refers to an event stream from the viewer statistics code in Listing 1, where a snapshot of each channel's viewers is published every 15 seconds. As in the Java implementation, channels having less than 2000 viewers are filtered out before they enter the window in order to prevent channels with only a few or no viewers from triggering drop events. The average is calculated from the events kept in the 1 minute window, while events older than this leave the window.

Listing 2 EPL Annoyance Detector

```
select channelName, viewers, avg(viewers)
  from ZapSnap(viewers > 2000).win:time(1 min)
  group by channelName
  having viewers < avg(viewers) * 0.85
```

5. EVALUATION

The main goal of this paper is to evaluate two paradigms for developing event-based systems, and specifically if it can be applied to our enhanced high-volume use case. Moreover, in this section we first give a brief analysis of the data obtained from the initial deployment of ZapReporter. This will be followed by a performance benchmark and software complexity evaluation.

5.1 Brief Data Analysis

To be able to predict the kind of traffic one might expect, when scaling up the number of events that will be generated, we examine the current trend of channel zapping. Hence, we selected a 15-day period (January 31 — February 14) from our logged datasets obtained using our current infrastructure, as described in Section 3.1. This period constitutes approximately 1.7G bytes of data, or 118M bytes per day. The sampled dataset contains events from 253,985 unique STBs, and 183 different channels were visited at least once during the period.

The number of events generated each day is shown in Figure 3, and the same data is also shown in Figure 4(a) sampled at hour intervals. An interesting observation from Figure 3 is that Wednesdays (5,12) and Thursdays (6,13) represent a significant deviation from average zapping activity. We speculate that this might be due to poor programming on these days across the board among broadcasters. In Figure 4(b), we show the distribution of zap events over a 24-hour period based on data from January 31. The plot confirms what is expected from habitual patterns, with a peak in zapping activity around 20:00. We leave it for future work, to provide an in-depth analysis of these data, when we have better accuracy.

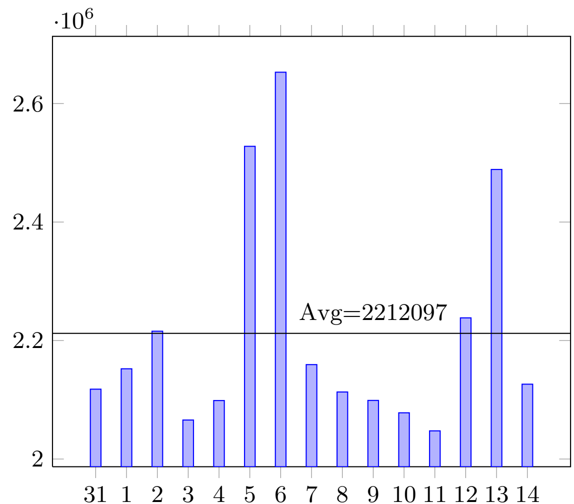
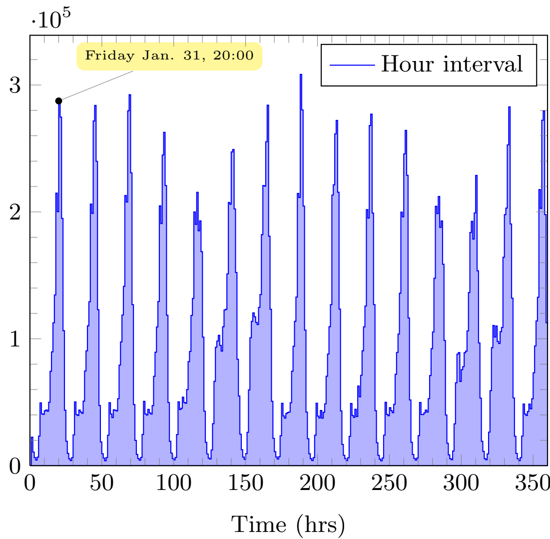
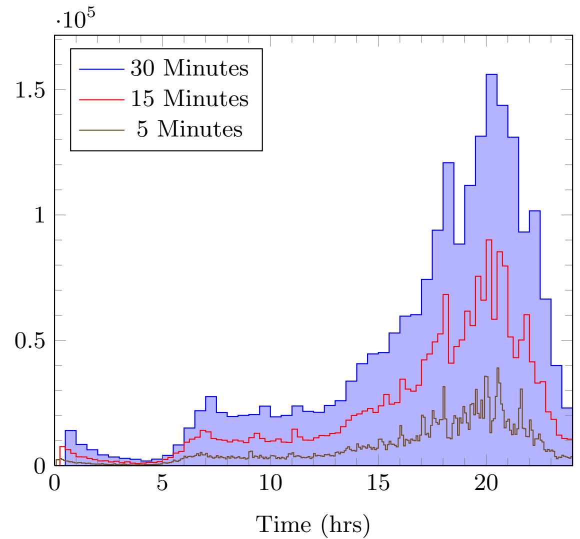


Figure 3: Number of zap events/day over a 15-day period.



(a) Number of zap events/hour over a 15-day period.



(b) Number of zap events over a 24-hour period.

Figure 4: Zap events observed using different sampling intervals.

5.2 Performance Evaluation

Here we provide a brief performance evaluation of both our implementations for the viewer statistics application. The annoyance detector application was difficult to test with Esper due to lack of real data. We are working on ways to simulate this also for this application.

5.2.1 *Environment and Experiment Setup*

To benchmark our applications, we used a server with RedHat Enterprise Linux 6, 64-bit, 14GB RAM, and a single Intel Xeon E5530 (8MB cache) Quad Core 2.4GHz CPU.

Since the current ZapReporter is not generating events at the desired rate, we wanted to verify if our implementations could sustain the expected traffic volume. Therefore, we built a test framework, in which we process a log file containing zap data from one day (January 31), carrying a total of 2,117,897 zap events. We measure the throughput obtained and memory usage while processing this file. The throughput is measured in four ways: (i) by reading the entire file into memory before processing it from memory, (ii) by reading the file line-by-line from disk, (iii) by receiving the events over UDP and (iv) by receiving the events via the HornetQ message bus. The reason for running both experiment (i) and (ii) was to reveal whether the performance bottleneck is I/O or CPU bound.

Each experiment was repeated 11 times, allowing one iteration for the Java hotspot compiler to optimize the code. The experiment results are presented as the average over ten iterations of each test, as shown in Figure 5. The results were validated by comparing the final state of both implementations, as they should end up with the exact same number of viewers per channel, and number of STBs observed after a completed run.

VisualVM v1.3.2 with a tracer plugin for collecting heap memory usage, was used to measure memory consumption. The sample rate was only 1 Hz, so the precision is limited, but nonetheless gives an overall impression of the memory consumption of the two implementations.

5.2.2 *Results*

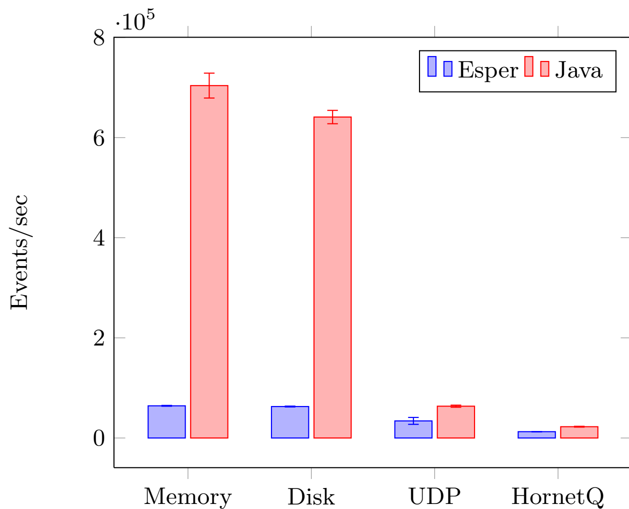
As seen in Figure 5(a), the native Java implementation outperforms the EPL implementation by a very large margin, with an average throughput surpassing 700,000 events per second compared to an average of only 64,275 events for the EPL version with the in-memory tests. Similar results are observed for the from-disk tests. We believe this can be accredited to the flexibility offered by a general-purpose language like Java to express and optimize data structures for the specific problem at hand. Relying only on pure EPL code to express complicated queries seems to hurt performance in a significant way.

Another interesting observation is the negligible performance hit on both implementations introduced by reading the events from disk instead of memory, indicating that the performance bottleneck is CPU-bound. By looking at Figure 5(a), it is also clear that receiving events over UDP introduces a significant performance penalty, reducing throughput by approximately 90 % for the Java implementation, from an average of 641,112 events per second (from-disk) to 63,515 events per second (UDP). Using the HornetQ message bus for event passing, a further performance hit is observed, to 22,546 events per second, or only 3.5 % of the throughput compared to reading the events from disk. For the EPL version, the throughput drops from 62,846 to 34,146 events per second (46 % reduction) over UDP, and to 12,623 events per second when using HornetQ.

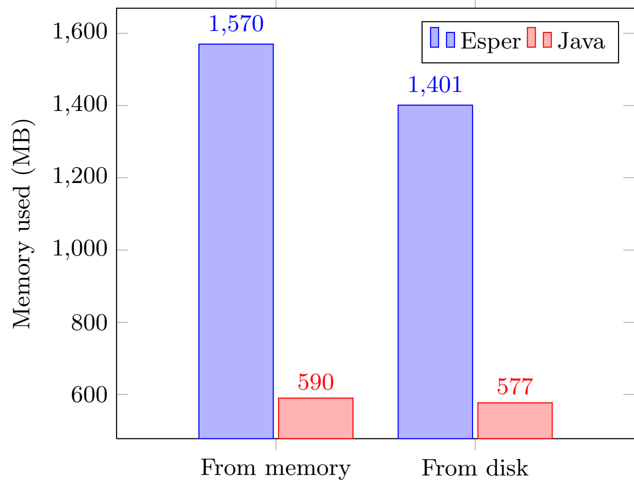
Although the performance hit on the Java application seems significant for the UDP and HornetQ cases, it still offers roughly 45 % higher throughput compared to the corresponding EPL versions. Moreover, the CPU load with the Java version is significantly lower.

The error bars in Figure 5(a) represent the standard deviation for each experiment. In the UDP experiments, the average packet drop for the Java version was 0.16 %, and 0.3 % for Esper. No packets were dropped by HornetQ.

Figure 5(b) once again shows the superior performance of the Java implementation. Average heap memory consump-



(a) Average throughput.



(b) Average heap memory consumption over 10 runs.

Figure 5: Throughput and memory performance.

tion of the Esper implementation is almost three times more than its Java counterpart, while it seems to confirm the negligible difference in performance between reading the events from disk versus loading them into RAM before processing.

5.3 Complexity

Software complexity is in general an equally important evaluation criteria to performance, when comparing the different approaches. Simpler code amounts to more robust and maintainable software [13], while the performance of hardware increases steadily. Therefore, we also evaluate our rather simple code examples using Halstead's software complexity metric along with a subjective discussion.

Complexity is measured using Halstead's formula [13, 21, 24, 2], that, when applied to the number of operators and operands in a program, is said to predict the following attributes:

- Length, volume, difficulty, and level of abstraction
- Effort and time required for development
- Number of faults

Predicting something that has already occurred is obviously self-contradictory, as the program must be developed before the number of operands and operators can be counted. The first two bullets are therefore in practice only used to validate the theory, and to give a metric of the complexity of a program, which is how it will be used in this evaluation.

There has been some dispute [12] regarding the usefulness and predictive powers of the Halstead metrics, and it could also be argued that the validity of these metrics are limited when applied to modern day object-oriented programming languages like Java, as they were conceptualized in an era of procedural languages. Nevertheless, we will include the non-predictive metrics, since these, together with total lines of source code, hopefully can give us some objective insight regarding the scope and complexity level of the implementations.

Originally, we used a software tool to automatically compute the Halstead metrics of the Java implementation. However, since we were unable find a tool that can compute the metrics for both Java and EPL, and because there are no universal consensus on the exact way of counting operators and operands in a given block of code [3], it was decided to calculate them manually instead, in order to ensure that the counting strategy is consistent between the two implementations.

Li *et al* [7] addresses some of the challenges involved in applying Halstead to object-oriented languages, and the essence of their findings is implemented in our own strategy for counting operators and operands. This includes ignoring import statements and package declarations, but counting everything that is necessary to express the program. Operators that are syntactical identical, but semantically different through context, are counted as different operators. Examples include the parenthesis '`()`' operator, which is counted as an operator in the case of grouping expressions, e.g. `(2+2)*4` and type casting, but not when used in methods. Furthermore, the dot operator '`.`' were ignored in package names when referring to objects, such as `tv.ChannelZap`, and included when delimiting an operator from an operand, as in `ZapWindow.std:unique()`. The colon operator '`:`' is also ignored in cases like this, when used to reference methods from package names, but included in statements like: `fields.hasNext() ? fields.next() : "OFF";`

Because Halstead's metric is designed to measure algorithms as opposed to complete programs [21], the metrics were calculated on class level in the Java implementation and subsequently summed together.

Figure 6 gives a break down per function for both EPL (upper bar) and Java (lower bar) implementations. It should be read as follows: The metric for the viewer statistics is shown to the left, followed by the metric for the annoyance detector application. In the case of Java, these are the only metrics necessary to represent both applications; event parsing is included in the code for the viewer statistics application. For the EPL implementation, we also include metrics for the additional Java code necessary for parsing,

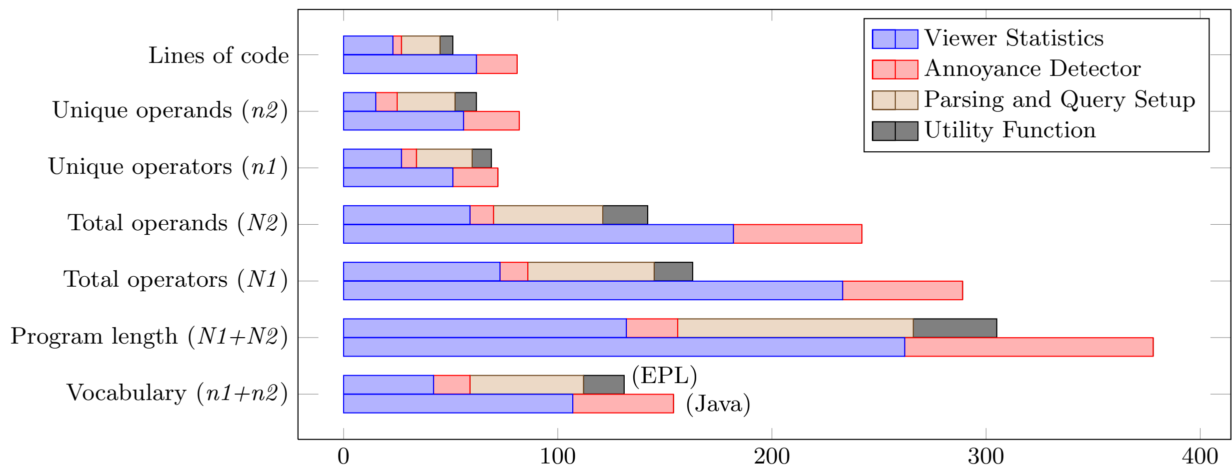


Figure 6: Complexity metrics break down per function for EPL (upper bar) and Java (lower bar).

Esper setup, and a custom utility function for calculating percentage. These are in addition to the query language itself. For both Java and EPL, the annoyance detector application builds upon the viewer statistics application, thus the numbers for the former includes the code from the latter.

On reading these metrics, it should be noted that the EPL implementation was done by a novice EPL programmer, and more efficient implementations might be possible.

The EPL implementation scores slightly better in all of the complexity metrics for the viewer statistics application, and significantly better for the annoyance detector. We do not find the difference in score between the two viewer statistics implementations wide enough to draw the conclusion that one is easier to develop than the other. However, upon expanding the basic viewer statistics application with annoyance detection capabilities, the additional programming effort required for expanding the Esper implementation (four lines of EPL) is significantly smaller than for the Java version (19 additional lines of code). The observed program length numbers points in the same direction, with an added program length of 116 versus only 24 for the EPL version.

One aspect of complexity, not covered by the software metrics, is the challenge of learning and understanding a new query language such as EPL. Although prior knowledge of SQL, possessed by many programmers, will be of great aid to this task. One concern in terms of using EPL for our applications is that we still had to write Java code to interface with other application code. Although, this interface code was minor in our case, it is easy to imagine having to write substantial amounts of wrapper/interface code outside of EPL for a variety reasons. Hence, it is obviously a disadvantage having to know and use two languages in order to develop an application. And another disadvantage with any declarative language is that we lose type-safety, an important software engineering principle for building robust applications.

Based on these observations, it is tempting to draw the conclusion that a general-purpose language is the most efficient tool for doing event stream processing. However, although it is the most effective implementation for the presented application in this case, there is reason to believe that dedicated event processing languages becomes more efficient relative to general-purpose languages upon expansion of the processing tasks, as indicated by the lesser effort required to add annoyance detection capabilities. This however, assume that streams can be reused across applications.

6. RELATED WORK

In their 2008 study, Cha *et al* [5] captured the channel changes of 250,000 households over a period of six months. By thoroughly analyzing this massive data set, the authors were able to create more accurate statistics of user behavior than with traditional sampling methods like the ones utilized by Nielsen Media Research [18]. This work is closely related to ours, in that it analyzes channel changes from a large IPTV network, but is strictly a statistical analysis, and does not consider any real-time applications that use the data.

Sripanidkulchai *et al* [22] performed an analysis of the live workload of a large content delivery network by analyzing data collected over a three month period, containing over 70 million requests. Like in our paper, they also identified flash crowds and usage patterns, but on audio and video streams delivered over the Internet, and not in a residential IPTV setting. Like Cha *et al*, this work does not deal with real-time pattern detection either.

Commercial vendors like JDS Uniphase [20], Mariner [16] and Agama [1] delivers agent-based solutions for monitoring Quality of Service (QoS) that also provides channel usage statistics. However, the interaction model of these solutions are all pull-based, either trough a SOAP API, graphical view from within the application, or through export functions that allows users to export historical data to a file. For the purpose of computing channel statistics and presenting them in near real-time, none of the commercially available solutions today have interaction models that is suitable for incorporating their functionality into a larger event-driven architecture. Moreover, they cannot be used to develop specialized applications like annoyance detection. The reasons for this can probably be attributed to business protectionism, attempting to lock IPTV operators to their solutions as much as possible, coupled with limited knowledge of the push-based interaction model that is vital in developing event-driven architectures and real-time functionality.

What separates this work from previous work is that none of the aforementioned solutions leverage event stream processing ideas to compute online channel usage statistics, limiting their use to identifying historical usage patterns and trends. By performing the computations online in near real-time, we are able to provide the users and operators with the added value of having instant access to emerging trends and usage statistics.

The other contribution of this work is the direct comparison between different paradigms for performing this type of event stream processing, which, to the authors' knowledge is the first of its kind.

7. CONCLUSIONS

In this paper, we have demonstrated that we are able to get much more accurate viewer statistics than with traditional methods by capitalizing on the two-way communication capabilities of IP-enabled STBs. By operating on the stream of zap events from STBs, we have been able to generate viewer statistics in two very different programming paradigms. Furthermore, our results show that the general programming paradigm outperforms the query language approach by a surprisingly wide margin for this fairly simple application scenario, while at the same time being fairly similar to its counterpart in terms of total lines of code (taking the additional required lines of Java code into account).

The debate of which paradigm to choose for a specific implementation should be about choosing the right tool for the job. If the application complexity is modest and performance requirements are high, it is probably more efficient to use a general-purpose language in most cases. If however the processing task at hand is very complex, and performance requirements are met with a more specialized language, going the query language route opens up possibilities for more effortless maintenance and expansion of the application at a later stage. It is probably wise to keep a generous performance margin in such cases, as our tests indicated that added complexity hurts performance of the more specialized tool more than its Java counterpart in applications like this, because of the limited flexibility in selecting appropriate data structures.

8. ACKNOWLEDGEMENTS

The authors would like to thank Ronny Lorentzen, Dagfinn Wåge and the IPTV team at Altibox for their valuable ideas, encouragement and helpfulness, and Bjarne Helvik for his assistance concerning Halstead's metrics.

9. REFERENCES

[1] Agama web site. Web, 2011. `http://www.agama.se`.

[2] B. Agarwal, S. Tayal, and M. Gupta. *Software Engineering & Testing: an Introduction.* Jones & Bartlett Learning, 2010.

[3] R. Al Qutaish and A. Abran. An Analysis of the Design and Definitions of Halstead's Metrics. In *15th Int. Workshop on Software Measurement (IWSM'2005). Shaker-Verlag*, pages 337–352, 2005.

[4] Altibox web site. Web, 2011. `http://www.altibox.no`.

[5] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatriain. Watching Television over an IP Network. In *IMC*, 2008.

[6] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Trans. Softw. Eng.*, 5:96–104, March 1979.

[7] V. Da Yu Li and O. Ormandjieva. Halstead's Software Science in Today's Object Oriented World. *Metrics News*, pages 33–41, 2004.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, January 2008.

[9] Esper Documentation. Web, 2011. `http://esper.codehaus.org/esper/documentation/documentation.html`.

[10] Esper, Performance-Related Information. Web, 2011. `http://esper.codehaus.org/esper/performance/performance.html`.

[11] Hva er TNS Gallup TV-panel? (What is TNS Gallup TV-panel?). Web, 2011. `http://www.tns-gallup.no/?aid=9072596`.

[12] P. G. Hamer and G. D. Frewin. M.H. Halstead's Software Science - a critical examination. In *ICSE*, 1982.

[13] B. E. Helvik. *Dependable Computing Systems and Communication Networks - Design and Evaluation.* Tapir academic publisher, January 2009.

[14] Latens web site. Web, 2011. `http://www.latens.tv`.

[15] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. In *SoCC*, 2010.

[16] Mariner Partners - IPTV Monitoring Software. Web, 2011. `http://www.marinerpartners.com`.

[17] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Advanced Event Processing and Notifications in Service Runtime Environments. In *DEBS*, 2008.

[18] Nielsen Ratings. Web, 2011. `http://en.wikipedia.org/wiki/Nielsen_ratings`.

[19] D. Nyvik. CEP: Integrator and facilitator for pub/sub messaging. Technical report, December 2009.

[20] J. Schmitt. NetComplete Home Performance Management (PM). White paper, November 2009. `http://www.jdsu.com/ProductLiterature/netcompletehomepm_WP_sas_TM_AE.pdf`.

[21] V. Shen, S. Conte, and H. Dunsmore. Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *IEEE Trans. Softw. Eng.*, 9:155–165, 1983.

[22] K. Sripanidkulchai, B. Maggs, and H. Zhang. An Analysis of Live Streaming Workloads on the Internet. In *IMC*, 2004.

[23] TNS Gallup. Web, 2011. `http://www.tns-gallup.no`.

[24] H. Zuse. *A Framework of Software Measurement.* Walter de Gruyter & Co., Hawthorne, NJ, USA, 1997.