DEBS Grand Challenge: Using a Distributed Blackboard for Real-Time Event Processing

Charles W. Rapp
Chicago Board Options Exchange
Chicago, IL, USA 60604
rapp@acm.org

ABSTRACT

This paper describes how a distributed blackboard system is used to process the real-time events generated from a football match where the ball and players are tagged with radio frequency identifiers to track their location, velocity, and acceleration. The distributed blackboard system is based on eBus, an open source event bus project, and the Java concurrency package.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – *Distributed Programming.*

General Terms

Algorithms, Performance, Design.

Keywords

eBus, event bus, Java concurrency, blackboard architecture, real-time event processing, distributed events, DEBS Grand Challenge.

1. INTRODUCTION

The Distributed Event-Based Systems (DEBS) 2013 Grand Challenge focuses on high throughput, low latency real-time event processing. Specifically, the events are from a football/soccer match where radio-frequency IDs (RFID) are used to track the location, velocity, and acceleration of the game balls, players, and referee. There is one RFID on the player left and right legs and one RFID on each of the goalkeepers' hands. The football RFID events are generated every 500 microseconds while the leg and hand RFIDs are updated every 5 milliseconds. With four ball RFIDs (one ball in play, three in reserve), seven players per team, two goalkeepers, and one referee, there are over 15,000 events per second.

The challenge is to continuously track four queries:

1. Current player running intensity and aggregate player running intensity. The challenge specifies six running intensity levels (from stopped to sprint). As the player changes speed, the query outputs how long the player spent at the previous level. These running intensities are aggregated by time and distance at four different reporting periods.
2. Player location "heatmap". Player location is tracked on four different grid layouts and four different reporting intervals.
3. Individual and team ball possession. Report which player currently possesses the football and collect the individual ball reports into a team ball possession time report.
4. Shots on goal. Report when a shot on goal starts and which player initiated the shot, update the event as the ball continues on its way, and issue a final event when the ball is either deflected, goes out of play or enters the goal.

The challenge is to find a distributed event solution which takes the low-level RFID events and generates the continuous query results with high-throughput and low-latency. Such efficiency is gained by decomposing the queries into sub-queries so that calculations are performed once and shared with all queries needing that result. Problem decomposition increases throughput by replacing the need to transmit the high volume low-level events with lower volume, higher-level, analyzed events.

Given the data-centric nature of this challenge and the advantages of query decomposition, a blackboard architecture was chosen for implementing the DEBS 2013 Grand Challenge solution.

2. BLACKBOARD ARCHITECTURE

The blackboard system architecture was first used by the Hearsay-II Speech Understanding System. [1] The system consists of three components: blackboard, knowledge sources (KS), and control The blackboard is a shared memory containing the posted events. It is divided into multiple levels arranged from low at the bottom to high at the top. The knowledge sources consist of three parts: triggering input events, condition, and action. The trigger specifies the event types required by the condition and action. When the required trigger events are posted to the blackboard, these events are passed to the condition and, if the condition is satisfied, the same events are passed to the action. The action will then post one or more new events to the blackboard, starting the knowledge source trigger cycle over.

The blackboard control is the central component to the blackboard system. The control is responsible for taking newly posted events and determining which (if any) knowledge sources are triggered by the event, choosing which triggered knowledge sources to test, and executing the triggered knowledge source conditions and actions.

The next section describes how the Grand Challenge queries are decomposed into Blackboard knowledge sources.

3. SOLUTION

The Grand Challenge solution is divided into the logical blackboard design and the physical host layout running the blackboard software.

3.1 Blackboard Design

The RFID sensor and referee events are posted to the blackboard level 0 by the Playback knowledge source. Because the challenge uses data collected from a football match, a mechanism is required to play the timestamped events back as close to real-time as feasible. The Playback KS also posts timer events at various intervals for use by the higher level knowledge sources responsible for reporting events at a fixed interval.

```
Blackboard Level 0:
  Playback KS:
    Trigger: None
    Condition: None
    Action: Post sensor, referee and timer events
```

Two knowledge sources trigger off the sensor events and post results to level 1: Ball KS and Player KS. There are four footballs in and around the field and all four balls are reporting their current position. The Ball KS determines which ball is actually in play and reports its sensor data in a separate ball event. This event contains a flag which specifies when the active ball has just gone out of play. The Ball KS reports no further ball events until an active ball is again determined. This knowledge source needs to deal with the situation where inactive footballs are re-distributed around the field by rolling balls across the field. In one instance, an inactive ball crosses the goal line at 48:28 into the game.

The Player KS collects the left and right leg sensor events for a given player and reports them together as a player event. This event contains the value-added information regarding the player's position, velocity, and acceleration based on the average of the two leg sensors. The player event also contains the hand sensors for goalkeepers. There is a Player KS instance for all sixteen players and the referee.

```
Blackboard Level 1:
  Ball KS:
    Trigger: Ball sensor events.
    Condition: Is active ball or in bounds?
    Action: Post ball event.

  Player KS:
    Trigger: Leg or Hand sensor events.
    Condition: None
    Action: Post player event.
```

The simplest Grand Challenge queries are at level 2: running intensity and heatmap. Both knowledge sources trigger on the player event. The running intensity KS tracks when a player changes running intensity and reports how long the player spent at the previous intensity. There are sixteen running intensity KS instances, one for each player.

```
Blackboard Level 2:
  Running Intensity KS:
    Trigger: Player event.
    Condition: Change in running intensity
    Action: Post running intensity event

  Heatmap KS:
    Trigger: Player or Timer events
    Condition: Has player changed cells?
    Action: Post heatmap event

  Kick KS:
    Trigger: Ball event
    Condition: Has ball meet acceleration minimum?
    Action: Post kick event

  Goal KS:
    Trigger: Ball event
    Condition: Has bell entered a goal?
    Action: Post goal event
```

The heatmap KS imposes a grid layout over the field and tracks which heatmap cell a player is currently occupying. On each one second timer event, the KS posts a heatmap event containing the percent time the player spent in each cell. There are 256 heatmap KS instances: one for each player (16), one for each heatmap grid (4 grid sizes), and one for each reporting interval (4 intervals).

The second level contains two helper knowledge sources: Kick KS and Goal KS. Both subscribe to the level 1 ball event. The Kick KS determines when the ball was kicked and the Goal KS decides if and when the ball entered a goal. There is one Kick and Goal KS instance.

The level 3 knowledge sources begin the event aggregation: aggregate running intensity and individual ball possession. The aggregate running intensity collects the individual running intensity events, tracking the time and distance spent at each intensity level. The aggregate running intensity event contains these totals for each intensity level and is posted with each new individual running intensity event and at interval end. There is an aggregate running intensity KS for each player (16) and reporting interval (4 intervals) for 64 instances in all.

The ball possession knowledge source tracks all players' current position and when the active ball is kicked. Ball possession is assigned to the player nearest to the ball and an initial ball possession event is posted. The knowledge source then triggers on further kick or ball events to determine if another player has gained possession or the ball went out of bounds. If so, a final ball possession event is posted to notify that the specified player has lost possession and the total possession time. There is only one ball possession knowledge source since it must track all 16 players in order to assign possession to one.

```
Blackboard Level 3:
  Aggregate Running Intensity KS
    Trigger: Running intensity event
    Condition: None
    Action: Post aggregate running intensity event

  Ball Possession KS
    Trigger: Player or Kick or Ball event
    Condition: None
    Action: Post ball possession event
```

The top and fourth blackboard level contains team ball possession and shot-on-goal queries. The team possession knowledge source tracks total possession time for each team and reports each team percent time in possession based on the total possession time for both teams. There is only one instance since the knowledge source must track total possession time for both teams in order to generate the team possession event.

```
Blackboard Level 4:
  Team Possession KS
    Trigger: Ball possession event
    Condition: None
    Action: Post team ball possession event.

  Shot-on-goal KS
    Trigger: Ball or Ball possession event
    Condition: Ball headed towards goal or
               Ball out-of play
    Action: Post shot-on-goal event
```

The shot-on-goal is the most sophisticated query. The KS looks for an initial ball possession event with the ball headed toward the opposition goal. If this is the case, an initial shot-on-goal event is posted. The KS then tracks the ball position, determining if the ball is still headed towards the target goal. If so, then an update shot-on-goal event is posted. If the ball's trajectory was deflected away from the goal or the ball went out of play, then a final shot-on-goal event is posted. There is only one shot-on-goal knowledge source instance since there is only active ball.

3.2 Host Layout

There are total of 360 knowledge sources instances which need to be distributed across multiple computers so as to minimize latency and still not overwhelm the computational resources.

```
Host 0:
  Knowledge Sources: Playback KS (1)
                     Ball KS (1)
                     Player KS (17)
                     Kick KS (1)
                     Goal KS (1)
                     = 21 instances
  Connections: None
Host 1:
  Knowledge Sources: Ball Possession KS(1)
                     Team Poss. KS (1)
                     = 2 instances
  Connections: Host 0
Host 2:
  Knowledge Sources: Running Instensity KS (8)
                     Agg. Run. Intensity KS (32)
                     = 40 instances
  Connections: Host 0
Host 3:
  Knowledge Sources: Running Instensity KS (8)
                     Agg. Run. Intensity KS (32)
                     = 40 instances
  Connections: Host 0
Host 4:
  Knowledge Sources: Heatmap KS (128)
                     = 128 instances
  Connections: Host 0
Host 5:
  Knowledge Sources: Heatmap KS (128)
                     = 128 instances
  Connections: Host 0
Host 6:
  Knowledge Sources: Shot-on-goal KS (1)
                     = 1 instance
  Connections: Host 0, 1
= 360 Knowledge Source instances, 7 connections.
```

Figure 1: Knowledge Source Distribution across Hosts

Host 0 contains the knowledge sources which process the high volume, high frequency sensor events, precluding the need to transmit those events across the network. The Ball KS alone reduces ball sensor events by 75% because only one of the four balls is in play. The Player KS makes no attempt to throttle sensor updates, posting a player event each time one of the player's sensors updates.

The remaining knowledge sources are distributed across six hosts prevent overwhelming the computational resources of any one host. These six hosts are dependent on Host 0 for triggering events - a situation which cannot be avoided since Host 0 is playing back the low-level game events. This is why it is critical for Host 0 to transmit events efficiently. This requirement plays an important role in designing a distributed blackboard system.

4. DISTRIBUTED BLACKBOARD

The Blackboard Architecture was designed with an eye towards parallel and distributed processing, it was unclear how blackboard problem-solving would be distributed [2]. Filman and Friedman [3] in 1984 stated that the centralized, global blackboard was the most difficult component to distribute. In 2013, distributing the blackboard events is easily accomplished using existing technologies: message-oriented middleware or a TupleSpace [4] to name two. Distributing the knowledge sources is also readily accomplished simply by running the application on multiple hosts.

The problem is the Blackboard control. Its role was originally designed to perform event distribution and task execution. But event distribution can be performed by middleware and task execution packages are part of Java [5] class libraries. So what other role is left for the Blackboard control?

The control is also responsible for keeping the Blackboard execution within computational resource and time limits. Blackboard systems were developed to run on a single computer with a fixed run time limit. Further, the Blackboard system was working on complex problems spaces like sonar signal analysis [6] and protein folding [7]. Given the large number of possible solutions to the problems, the number of triggered knowledge sources is also large. If all triggered KSes were tested, it would overwhelm the computational limits - memory and/or processor time exceeded.

The control has the duty of ranking triggered knowledge sources from most likely to lead to a correct solution down to those least likely. The control then tests and executes only the most likely triggered KSes. This ranking is feasible in a non-distributed environment because the control knows all the triggered knowledge sources. But a distributed control knows only about the locally triggered KSes. It may be that running the locally best triggered KSes is acceptable globally, but if a global ranking is necessary, then a distributed control is needed. Developing a distributed Blackboard control implementing global control is the focus of much research over the years. [8] [2]

But is a control necessary in all circumstances? We are now in a world with hyper-threaded, multi-core, multi-processor computers with multi-gigabyte memory, terabyte solid-state drives. and communicating across gigabit networks. It would be take a complex problem space to exceed these limits.

Dropping the control makes the Blackboard Architecture easily distributable. The knowledge sources receive triggering events from an event distribution system and, when the trigger set is complete, use a thread pool to test the KS condition and perform the KS action. This section describes how a distributed Blackboard system was implemented using the eBus event bus and the `java.util.concurrency` [9] package.

4.1 An eBus Overview

eBus [10] is a broker-less message distributor supporting advertise/subscribe and request/reply paradigms, providing space and synchronization decoupling [11]. eBus operates at the object level, delivering messages to an object rather than to a process. The eBus API is available in both Java and .Net. The remainder of this section focusses on advertise/subscribe event distribution using the Java API.

eBus messages are defined as a class extending the `net.sf.eBus.messages.EMessage` abstract class. All message fields must be public, read-only, and a data type which eBus can serialize. This includes Java primitives, `java.lang.String`, `java.util.Date`, Java `enum`, a type implementing `net.sf.eBus.io.ESerializable`, and arrays of these types. The `@EMessageInfo` class annotation is required which associates a unique 8-byte, signed integer identifier with the class, provides the message type (notification, request, reply, or system) and the ordered list of message fields. While this last attribute seems redundant given that the message fields can be retrieved through reflection, the list also defines the field serialization order.

```
@EMessageInfo(messageId=SensorEvent.MESSAGE_ID
             messageType=MessageType.NOTIFICATION,
             messageFields={"ts", "sensor"})
public final class SensorEvent extends EMessage
{
    ...
    public final long ts;
    public final Sensor sensor;
    public static final long MESSAGE_ID= 1000L;
}
```

Figure 1: Challenge Sensor event

A message is identified by a message key: a tuple containing the message class (both the Java class instance and the unique integer identifier) and string subject. eBus uses both type and topic to connect subscribers with publishers. [11] eBus subjects are free-form, allowing the application to define a naming scheme such as used by topic-based addressing systems like TIBCO Rendezvous [12].

This scheme separates the subject from content, allowing a subscriber to choose different content for the same subject. For example, it is natural to use the same subject "AAPL" for accessing different market data messages `Trade`, `Quote`, and `Closing`. Subject-based routing requires three different subjects (e.g. AAPL/trade, AAPL/quote, and AAPL/closing) to differentiate. The message key is used to link publisher to subscriber using an advertise/subscriber paradigm.

A publisher is a class which implements `EPublisher` and advertises the message keys it can publish. The advertised message key subject may be a regular expression (e.g., `/Sensor/ball[1-4]`), allowing the publisher to match multiple subscriptions. Once eBus accepts the advertisement, the publisher is still not ready to publish until there is a subscriber to the event.

```
public interface EPublisher {
    void publishStatus(boolean upFlag,
                       EAdvertisement ad,
                       ESubject subject);
}
```

Figure 2: eBus Publisher Interface

A subscriber class implements `ESubscriber` interface and subscribes to the desired event message key. The subscriber must use a concrete message key subject (e.g., `/Sensor/ball2`) because eBus does not support multi-subject subscriptions.

eBus looks up the `ESubject` instance using the subscription message key in a Ternary Search Tree [13], adding a subject if not found. `ESubject` is the heart of eBus event distribution; maintaining the set of all extant publishers and subscribers. When the first subscriber is added to the subject, the subject calls the `EPublisher.publishStatus(true, ad, this)` (where ad is the token returned by the eBus advertise method and this is the `ESubject` instance) to inform the publisher that it is clear to publish events on the subject. The publisher uses the `ESubject` instance to publish events and the subject instance then forwards the event to all subscribers via the `notify()` method. There is no routing involved in the event transmission because the publisher and subscribers were linked when the subscription was placed.[1]

[1] Message key subject length and complexity is irrelevant in eBus unlike topic-based publish/subscribe where performance degrades with topic length and complexity. [12]

```
public interface ESubscriber {
    void feedStatus(boolean upFlag,
                    String reason,
                    int pubId,
                    int pubCount,
                    ESubscripton sub);
    void notify(EMessage event,
                int pubId,
                ESubscription sub);
}
```

Figure 3: eBus Subscriber Interface

When the subject's last subscription is retracted, eBus calls `publishStatus(false, ad, this)` to stop the event publication. ESubject is also able to inform subscribers when there are no publishers for an event by calling `feedStatus(false, "no publishers", NO_PUB_ID, 0, sub)`, allowing the subscribers to take necessary corrective action.

eBus delivers events between publishers and subscribers existing in the same virtual machine or across a network. eBus publishers and subscribers are space decouple and their respective behavior is the same whether communicating locally or remotely. The only restriction is that eBus will not deliver an event received from a remote publisher to a remote subscriber. Either the publisher or the subscriber must be local to the process. The eBus API will not act as a router between applications.

4.2 Implementing a Distributed Blackboard

Without a control, the knowledge source itself is responsible for receiving posted events, determining if it is triggered, and posting resulting events back to the blackboard. In eBus terms, a knowledge source is both a publisher and subscriber.

```
public abstract class KnowledgeSource
    implements ESubscriber, EPublisher
{
    private final EMessageKey[] _subKeys;
    private final EMessageKey[] _adKeys;
    private final List<Trigger> _triggers;

    public abstract boolean condition(EMessage[]);
    public abstract void action(EMessage[]);

    protected KnowledgeSource(EMessageKey[] subs,
                              EMessageKey[] ads){
    }

    @Override
    public final void notify(EMessage msg,
                             int pubId,
                             EAdvertisement ad) {
        for (Trigger t : _triggers) {
            t.add(msg);
            if (t.isComplete() == true) {
                _threadPool.execute(
                    new KSTask(this, t));
            }
        }
    }
}
```

Figure 4: Knowledge Source as an eBus client

The abstract `KnowledgeSource` base class stores the subscription and advertisement message keys, performs the subscribing and advertising for the subclasses, and performs the inbound event handling. `KnowledgeSource` tracks all in-progress triggers and when a trigger set is complete (i.e., contains all required events of the proper type), a `KnowledgeTask` is created and added to a `java.util.concurrency.ExecutorService` execution.

```
public final class KSTask implements Runnable {
    private final KnowledgeSource ks;
    private final EMessage[] events;

    @Override
    public void run() {
        if (ks.condition(events) == true) {
            ks.action(events);
        }
    }
}
```

Figure 5: Triggered Knowledge Source Task

`KSTask` applies the trigger events to the knowledge source condition. If the condition is satisfied by the events, then the task applies the same events to the knowledge source action which may result in publishing further events.

Grand Challenge knowledge sources (like the running intensity KS) are classes extending the abstract `KnowledgeSource` class, providing the subscription and advertisement messages keys, and implementing the `condition` and `action` abstract methods. Because `KnowledgeSource` performs the work of receiving events and determining when the knowledge source is triggered, the application knowledge source class development focuses on problem solving.

This also shows how a distributed *uncontrolled* Blackboard architecture may be readily implemented when based on an existing event distribution system and thread pool library. A difficulty is that eBus does not provide historical event retrieval. Publishers and subscribers are *time-coupled*. [11] A subscriber receives only those events published after the subscription is placed. Therefore, the eBus-based Blackboard system must instantiate all knowledge sources and have the KS subscriptions in place prior to posting events to the Blackboard. This is accomplished by using a distributed “CountDownLatch” and when all KS instances report their readiness, a “blackboard start” event is posted which triggers the initial knowledge sources (in this case, the Playback KS) to begin posting low-level events.

5. RESULTS

Table 1 shows the number of events posted per type, per host. Host 0 generated some 47 million sensor events which the Ball and Player KSes reduced to 27.7 million ball and player events, respectively. Hosts 4 and 5 produced one half million heatmaps events each. Host 1 generated 183,588 individual and team ball possession events for the game. Hosts 2 and 3 each posted some 12,000 running intensity events and Host 6 posted 95 shot-on-goal events.

The total number of events posted for the entire game is 76,424,641. With a total game playing time is 3,597 seconds, this makes for 21,246 events posted per second.

(This paper, regrettably, does not contain any latency or throughput statistics as the author lost access to the necessary testing environment while developing this Grand Challenge solution.)

6. CONCLUSION

The first and most significant conclusion is that 21 thousand events per second is well within the capabilities of computers and networks, circa 2013. If all posted events were 512 bytes on the wire, that comes to 88 million bits per second, a rate well within a gigabit network link. Therefore, any claim to this solution’s low latency and high throughput cannot be attributed to the solution. Rather, it must be expected given that the sensor event rate does not tax computational resources.

Table 1. Event Counts per Host

Host	Event	Count
Host 0	Sensor	47,385,993
	Referee	3
	Timer	4,415
	Ball	5,703,759
	Player	22,029,552
	Kick	36,915
	Goal	11
Host 1	Ball Possession	20,380
	Team Possession	163,208
Host 2	Running Intensity	2,309
	Agg. Running Intensity	9,908
Host 3	Running Intensity	2,324
	Agg. Running Intensity	9,968
Host 4	Heatmap	528,000
Host 5	Heatmap	528,000
Host 6	Shot-on-goal	95
	Total	76,424,641

But this does explain why the uncontrolled, distributed blackboard system did work. This design only works when the problem space itself stays within set limits. The blackboard knowledge sources form a straight and short data pipeline from the initial sensor, referee, and timer events to the final query results. The knowledge sources, particularly the heatmap query, are memory-bound rather than CPU or network I/O bound. But this situation is common today where the time-space trade-off is now in favor of using more space to decrease time.

To re-iterate, the uncontrolled, distributed blackboard system cannot guarantee staying within set computation resource and time limits because it is uncontrolled. Control can be put back into the design by replacing the Java `ExecutorService` with a Blackboard control thread pool. This control would rank triggered knowledge sources and run the *local* maximum only, holding the minimum KSes in abeyance until there is a spare thread or dropping them if not run in within a set time window.

A variation would be to test all triggered knowledge sources if testing proves cheap enough and then use the control thread pool to prioritize the knowledge source actions as the actions are the most expensive knowledge source component.

But this control has only local knowledge and its selection of a locally best triggered knowledge source may prove to be a globally second-best choice. The solution used by Protean [7] was to use control knowledge sources which worked together to focus the blackboard system along the best solution path. Since they are knowledge sources like the problem-space KSes, the control KSes can use eBus to communicate remotely. This is a line of future research for a controlled, distributed Blackboard Architecture.

7. REFERENCES

[1] Erman, L. D., Hayes-Roth, F., Lesser, V. V., and Reddy, R. 1980. The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computing Surveys*. 12, 2 (June 1980), 213-253. DOI=http// doi.acm.org/10-1145/356810-356816.

[2] Corkill, D. D. 1989. Design alternatives for parallel and distributed blackboard systems. In *Blackboard Architectures and Applications,* V. Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, Ed. Academic Press, Inc., Orlando, FL. 99-136.

[3] Filman, R. E. and Friedman, D. P. 1984. *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw-Hill, New York, NY.

[4] Gelernter, D. and Carriero, N. 1992. Coordination languages and their significance. *Communications of the ACM*. 35, 2 (February 1992), 97-107. DOI= http://doi.acm.org/ 10.1145/129630.129635.

[5] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. 2006. *Java Concurrency in Practice*. Addison-Wesley Professional, Indianapolis IN.

[6] Nii, H. P., Feigenbaum, E. A., Anton, J. J., and Rockmore, A. J. 1989. *Readings from the AI Magazine*. American Association for Artificial Intelligence, Menlo Park, CA

[7] Hayes-Roth, B. 1985. A blackboard architecture for control. *Journal of Artificial Intelligence*. 26, 3 (July 1985), 251-321. DOI=http://doi.acm.org/10.1016/004-3702(85)90063-3.

[8] Lesser, V. R. and Erman, L. D. 1980. Distributed Interpretation: A Model and Experiment. *IEEE Trans. on Computers*. 29, 12 (Dec. 1980) 1144-1163. DOI=http:// doi.acm.org/10.1109/TC.1980.1675519.

[9] Java concurrency package http://docs.oracle.com/javase/7/docs/api.

[10] eBus http://ebus.sourceforge.net/eBus/Welcome.html.

[11] Eugster, P. Th., Felber, P. A., Guerraoui, R., and Kermarrec, A-M. The many faces of publish/subscribe. *ACM Computing Surveys*. 35, 2 (June 2003) 114-131. DOI=http://doi.acm.org/ 10.1145/857076.857078.

[12] TIBCO Rendezvous Manual http://docs.tibco.com/pub/rendezvous/8.3.1_january_2011/ html/tib_rv_concepts/wwhelp/wwhimpl/common/html/ wwhelp.htm#context=tib_rv_concepts&file=rv_concepts. 6.036.html

[13] Bentley, J. L. and Sedgewick, R. Fast algorithms for sorting and searching strings. In *SODA'97 Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms* (New Orleans, LA, Jan 4 - 7, 1997). Society for Applied and Industrial Mathematics, Phil., PA. 360-369.