Content-Based Publish/Subscribe with Structural Reflection

Patrick Th. Eugster *Rachid Guerraoui*
Distributed Programming Group
Swiss Federal Institute of Technology
Lausanne

{Patrick.Eugster, Rachid.Guerraoui}@epfl.ch

Context

- **DACE project: quest for**
  - Paradigms
  - Abstractions
  - And algorithms for distributed (large-scale) computing
  - Language integration
- **Distributed *interaction***
  - Publish/subscribe paradigm: information bus
  - Decoupling of participants in
    - Time
    - Space
    - Flow

Distr. Asynchronous Collections

- **Generalization of event channels, message queues, …**
- **Accessible from various nodes**
- **Essentially distributed**
  - No centralized component
  - Increased availability
- **Notification mechanism**
  - Subscribing is expressing interest in new elements
  - Observer design pattern: subscriber is observer
- **One size fits all**
  - Different QoS
  - Different variants according to different interaction flavors, e.g, push/pull

Publish/Subscribe

- **Topic-based (subject-based)**
  - News-like approach
    - Messages are classified according to topic names
    - Hierarchies, wildcards, aliases
  - Static, limited expressiveness
- **Content-based (property-based)**
  - Consumers subscribe by specifying *properties* of messages
  - Application criteria is seen as *pattern*
  - Pattern translated to *filter*, also seen as *predicate*
  - Dynamic, more difficult to implement

Approaches

- ***Properties* usually interpreted as *attributes***
  - Subscription pattern is described by attributes and expected values
- **Subscription language**
  - Specific grammar and parser
  - Mainly based on attributes
  - E.g., `"sender is bob"`
- **Template objects**
  - Runtime message objects compared to predefined objects
  - Comparison attribute-wise

Model

- **Looking for a *pragmatic* approach, respecting**
  - Encapsulation: *description* of properties and *filtering* not based on attributes
  - No subscription language, only language constructs
  - Any "serializable" object can be used as message object (no specific types)
  - Subscription pattern must not be opaque to middleware
- **Outline**
  - Subscription pattern uses method invocations for querying
  - Different comparison styles
  - Nested method calls increase expressiveness

Accessors

- **Represent a means to query a message object**
- **Characterized by**
  - A set of method/arguments pairs $(M_1, P_1), \ldots, (M_k, P_k)$
  - Represents invocation chain to access information
  - Sideffects not considered
- **Java Accessor interface**

```
public interface Accessor {
  public Object get(Object m) throws Exception;
}
```

- **Implemented by application**
- **General-purpose accessor `Invoke`**
  - Implemented with structural reflection: Java core reflection
  - M_i specified as `Method (java.lang.reflect)` objects
    - Type of message objects is known
    - Method object lookup only once (in application)
    - Explicit use of reflection
  - M_i specified by name (and signature)
    - Lookup for every message object
    - Structural conformance

Conditions

- **Represents a basic condition on message objects**
- **Characterized by**
  - An *accessor*
  - A predefined *result*
  - A *comparator* (comparison function)
    - Binary predicate
    - Can be seen as M_{k+1}
- **Evaluation**
  - The binary predicate compares the predefined result and
  - The result of the invocation chain

- **Condition interface**

```
public interface Condition {
  public boolean conforms(Object m); }
```

- **Implemented by application**
- **Library of conditions, varying by the comparison**
  - Comparator is method on one of objects, or `static` method
  - `Equals`: Java `equals()`
  - `Compare`: Java `compareTo()` (`Comparable`) for ordered types, e.g., `Integer`
  - Shortcuts (M_n), e.g., `isInstance()`

Patterns

- **Subscription pattern**
  - A set of conditions C_1, …, C_n and a function on those
- **Evaluation**
  - Every condition is evaluated
  - The function is evaluated
- ***F* is constructed by combining basic conditions**
  - Logical *and: And(m)=*C_1*(m) and* C_2 *(m)*
  - Logical *or: Or(m)=*C_1 *(m) or* C_2 *(m)*
  - *Xor*, …, and *Not*
  - Patterns are conditions

Programming Example

Event class

```
public class ChatMsg implements java.io.Serializable {
  private String sender;
  private String text;
  public String getSender() { return sender; }
  public String getText() { return text; }
  public ChatMsg(String sender, String text)
    { this.sender = sender; this.text = text; }
}
```

Create a local DAC proxy

```
DASet myChat = new DAStrongSet("/Chat/Insomnia");
```

Insert new objects (publish)

```
myChat.add(new ChatMsg("Bob", "Hi from Bob"));
```

Advertise interest in new objects (subscribe)

```
public class ChatNotifiable implements Notifiable {
  public void notify(Object m, String DACName) {
    System.out.println(((ChatMsg)m).getText()); }
}
myChat.contains(new ChatNotifiable());
```

Content-Based Subscribing

- **Step 1: verify if message comes from "Alice"**
  - Java `equals()` in `String`
- **Construct accessor (explicitly)**

```
Accessor getAlice = new Invoke("/getSender", null);
```

- **Construct condition**

```
Condition fromAlice = new Equals(getAlice, "Alice");
```

- **Subscribe**

```
myChat.contains(new ChatNotifiable(), fromAlice);
```

Step 2: verify if message text contains "Bob"

- Java `indexOf()` in `String:` if not contained `-1` returned

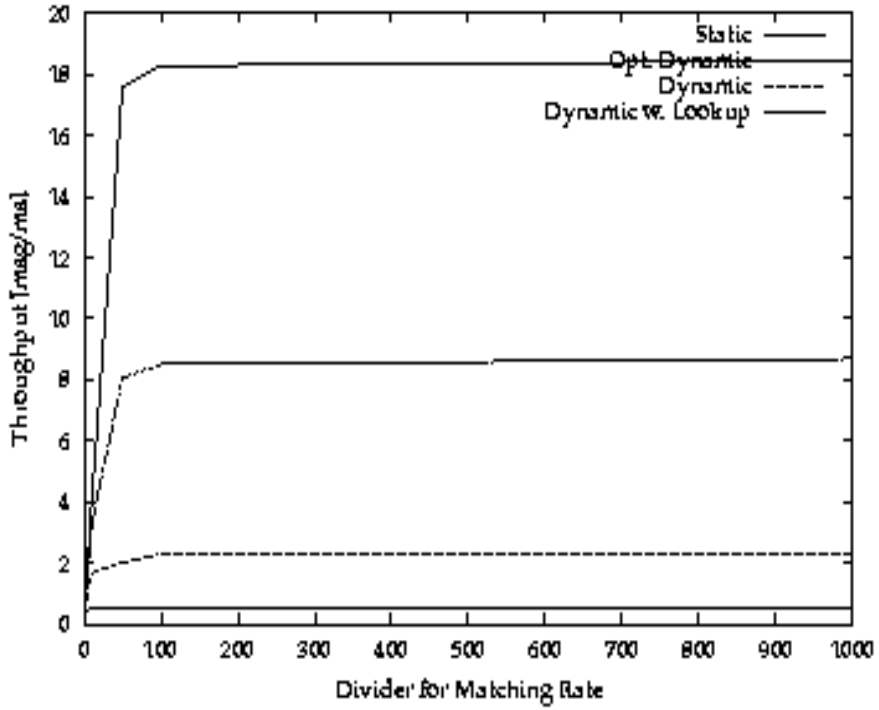
Construct pattern

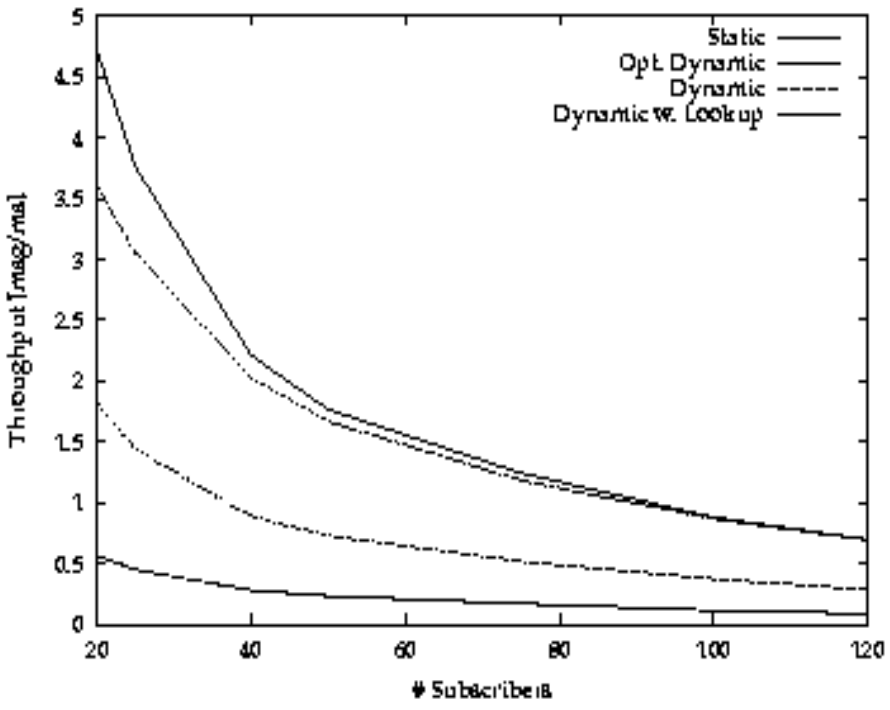
```
ExtendedCondition fromAlice =
  new Equals("/getSender", "Alice");
Object[] args = new Object[]{null, {"Bob"}};
ExtendedCondition noBob =
  new Equals("/getText/indexOf", args, new Integer(-1));
myChat.contains(new ChatNotifiable(),
  fromAlice.and(noBob.not()));
```

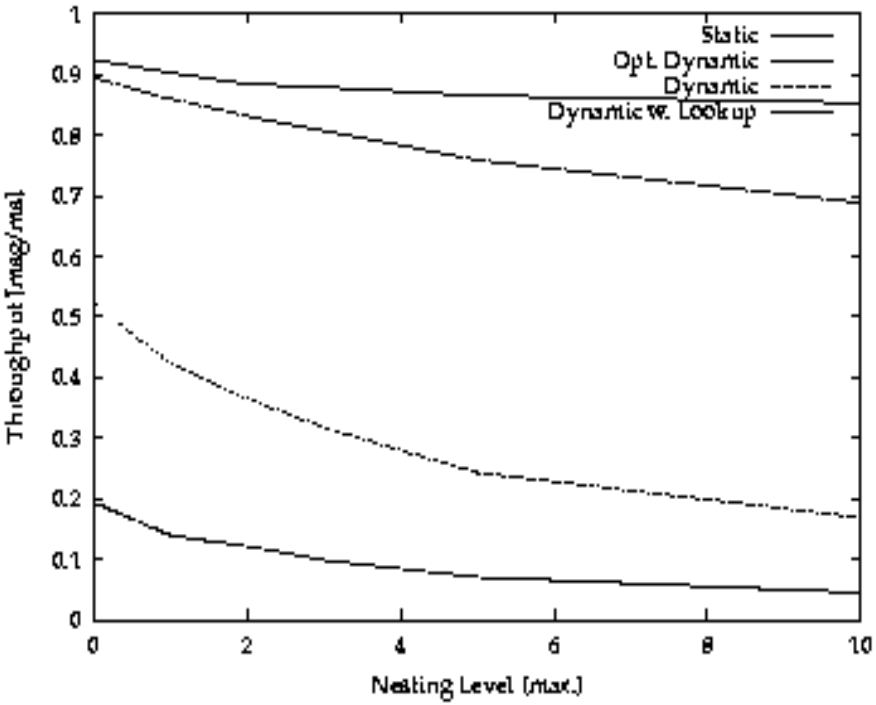
Optimizations

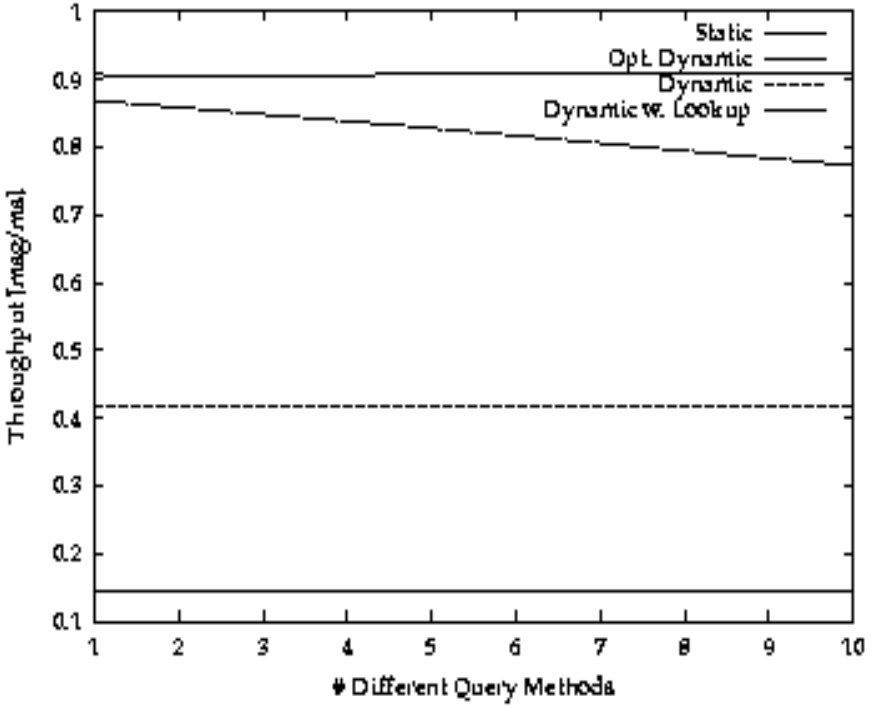
- **Application provides method *object***
  - Type of messages is known
  - Redundant invocations can be avoided
  - Static code can be generated
    - Type casts
    - Caution: primitive types
    - Uses `sun.tools.javac`
- **Application provides method *name***
  - Type of messages is unknown
  - Can be explicitly specified by name
  - *Type-based* subscription scheme implicitly adds knowledge

Performance






Static
Opt. Dynamic
Dynamic
Dynamic w. Lookup
Throughput [msg/ms]
Nesting Level [max.]
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2
4
6
8
10


Static
Opt. Dynamic
Dynamic
Dynamic w. Lookup
Throughput [msg/ms]
Different Query Methods
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1
2
3
4
5
6
7
8
9
10

Final Comments

- **Java reflection**
  - Type check arguments
  - Does the return type of M_i implement M_{i+1}?
  - Primitive types
- **Pragmatic approach**
  - Proof of feasibility
  - No extension of Java
- **Language integration**
  - Requirements for type-safe distributed interaction?
  - For pattern expression?