Linguistic Support for Distributed Programming Abstractions

Christian H. Damm
Microsoft Business Solutions
Vedb, Denmark

Patrick Th. Eugster
Sun Microsystems
Volketswil, Switzerland

Rachid Guerraoui
Swiss Federal Institute of Techn.
Lausanne, Switzerland

Context

- Distributed applications
  - Built on top of “middleware”
- Implementing (type safe) middleware
  - What abstraction?
    - E.g., remote procedure call, messaging, …
  - How to implement the “interface” between middleware and applications (depending on application types)?
    - I.e., stubs, proxies, adapters, connectors, …
    - E.g., precompiler, language-integration (compiler), library, …
- General-purpose programming language features for library implementation?

Roadmap

- Context
- Type-based publish/subscribe (TPS)
- TPS in a programming language
- TPS as a library
- A “futuristic” TPS library
- Comparison
- Conclusions

Publish/Subscribe

- Shared information bus, event channel
  - Multicast abstraction
  - Distributed components communicate indirectly, by
    - Publishing events (messages)
    - Subscribing to events (messages)
  - Decoupling of components

TPS in Short

- Motto
  - Events are objects, instances of application-defined types
- Publishing objects
  - A copy of a published object is created for each interested subscriber
- Subscribing to object types
  - No explicit „subject“, the type is the subject
  - No explicit „properties“, the state is the content
    - Subscriptions expressed as predicates on public members, i.e., fields and methods
    - Subscriptions should be „migratable“

Roadmap

- Context
- Type-based publish/subscribe (TPS)
- TPS in a programming language
- TPS as a library
- A "futuristic" TPS library
- Comparison
- Conclusions

TPS in a Programming Language

- Merging middleware and programming language
  - Investigate feasibility of TPS principles
- $Java_{PS}$
  - `publish new StockQuote(…);`
  - `subscribe (StockQuote q){…}{…};`
- Implemented with extended compiler
  - Heterogenous translation
  - Generation of “adapter code” e.g., `StockQuoteAdapter`
  - Invocations of primitives transformed, performed on adapters

Programming with Java$_{PS}$

```
public class StockQuote
   implements ... {
 private String company;
 private float value;
 private int amount;
 public String getCompany()  {
   return company; }
 public float getValue() {...}
 public int getAmount() {...}
 public StockQuote(String c,
                  float v,
                  int a)
   { company = c;
   value = v;
   amount = a;
 }
}
```

- Publishing stock quotes

```
StockQuote q = new
 StockQuote("Telco", 100.0, 25);
publish q;
```

- Subscribing to stock quotes

```
Subscription s =
 subscribe(StockQuote q)
{
 return
  q.getCompany().equals("Telco");
}
{
  System.out.println(q.getPrice());
}
s.activate();
```

Roadmap

- Context
- Type-based publish/subscribe (TPS)
- TPS in a programming language
- TPS as a library
- A "futuristic" TPS library
- Comparison
- Conclusions

TPS as an External Library

- First-class software bus or event channel
- Distributed Asynchronous Collections (DACs)
  - Variant of well-known collection abstraction

    ```
    interface DAC extends Collection {}
    ```

  - Intuitive use, integrated with inherent Java collection framework
  - Adding an element to a DAC comes to publishing that element
  - Browsing a DAC for particular elements expresses an interest in these elements, and is interpreted as subscription

Programming with DACs

- Publishing stock quotes

```
DAC qs =
  new DASet("StockQuote");
StockQuote q =
  new StockQuote("Telco",
                 100.0,
                 25);

qs.add(q);
```

- Subscribing to stock quotes

```
class MySubs
  implements Subscriber {
 public void notify(Object o) {
  StockQuote q = (StockQuote)o;
  System.out.println(q.getValue());
 }
}

Accessor acc =
  new Invoke(".getCompany", null);
Condition myCond =
  new Equals(acc, "Telco");
qs.contains(new MySubs(), myCond),
```

Roadmap

- Context
- Type-based publish/subscribe (TPS)
- TPS in a programming language
- TPS as a library
- A “futuristic” TPS library
- Comparison
- Conclusions

A Futuristic TPS Library

- Application-defined event types and type safety
  - A first-class channel abstraction must comply to the event type
- GDACs are parameterized by event type

```
interface GDAC<T> extends Collection<T> {
  void add(T t);
  Subscription<T> contains(Subscriber<T> st);
  ...}
```

- Generic Java (GJ)
  - Parametric polymorphism (F-bound polymorphism)
    - A „future“ version of the Java language (1.5)
- **`Subscription`** enables the expression of predicates
  - Behavioral reflection introduced in Java 1.3 (dynamic proxies)

Programming with GDACs

- Publishing stock quotes

```
GDAC<StockQuote> qs =
  new GDASet<StockQuote>();
StockQuote q =
  new StockQuote("Telco",

                               100.0, 25);

qs.add(q);
```

- Subscribing to stock quotes

```
class MyStockSubs
  implements Subscriber<Stockquote>
{
  public void notify(Stockquote q) {
   System.out.println(q.getValue());
  }
}

Subscription<StockQuote> s =
   qs.contains(new MyStockSubs());
Stockquote q = s.getProxy();
q.getCompany().equals("Telco");
s.activate();
```

Roadmap

- Context
- Type-based publish/subscribe (TPS)
- TPS in a programming language
- TPS as a library
- A “futuristic” TPS library
- Comparison
- Conclusions

Comparison

- Simplicity: programming effort
  - 1. JavaPS, 2. GDACs, 3. DACs
- Flexibility: extension effort
  - 1. GDACs/DACs, 3. JavaPS
- Type safety: deployment
  - 1. JavaPS/GDACs, 3. DACs
- Performance: overhead
  - Negligible overhead of reflection
  - No overhead for genericity (type casts inserted at compilation)

Shortcomings

- No runtime support for genericity

```
GDAC<StockQuote> qs =
   new GDASet<StockQuote>(StockQuote.class);
```

- No dynamic proxies for classes

```
Stockquote q = s.getProxy();
q.getCompany().equals("Telco");
```

- No dynamic proxies for primitive types

```
Stockquote q = s.getProxy();
q.getValue() < 100.00;
q.getCompany().equals("Telco") && ...;
```

Roadmap

- Context
- Type-based publish/subscribe (TPS)
- TPS in a programming language
- TPS as a library
- A “futuristic” TPS library
- Comparison
- Conclusions

Conclusions

- Genericity
  - For type safe „interface“ abiding to applications
  - Should include runtime support
- Reflection
  - For implementation behind „interface“ on top of untyped network
  - Should include structural and behavioral reflection
- Type system
  - Should be simple and uniform, e.g., no hybrid type system
- TPS stringent demands
  - Requirements include many other abstractions
- .NET?

Questions