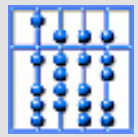
Ubiquitous Tracking using the DWARF Middleware

Ubiquitous Tracking Project Workshop

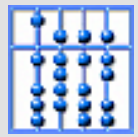
Asa MacWilliams
Lehrstuhl für Angewandte Softwaretechnik
Institut für Informatik, Technische Universität München
macwilli@in.tum.de

Feb 6 2004



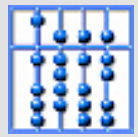
Summary

- An implementation of the Ubiquitous Tracking concepts needs middleware infrastructure
- DWARF contains decentralized, adaptive middleware which is well-suited to this task
- The DWARF middleware can contribute to:
  - Communication between software components
  - Discovery of new devices in environment
  - Configuration and adaptation of components
  - Formation of data flow networks
- However, it will need to be extended for
  - Scalability
  - Performance
  - Ad hoc networks



Motivation

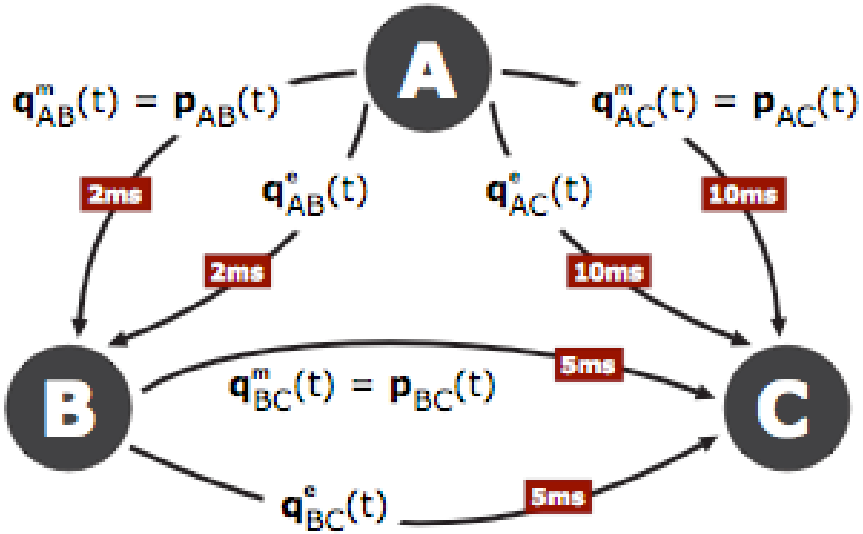
- Within the Ubiquitous Tracking project:
  - Distribution is part of the game
  - Ad hoc discovery and configuration of trackers
  - Formation of data flow graph
- Within the DWARF project:
  - For ubiquitous AR, we need ubiquitous tracking
  - Using DWARF in lets us leverage existing components
- My personal motivation:
  - DWARF architecture and middleware are basis of my Ph.D. thesis
  - Ubiquitous tracking is a good application to “harden” the framework and the middleware

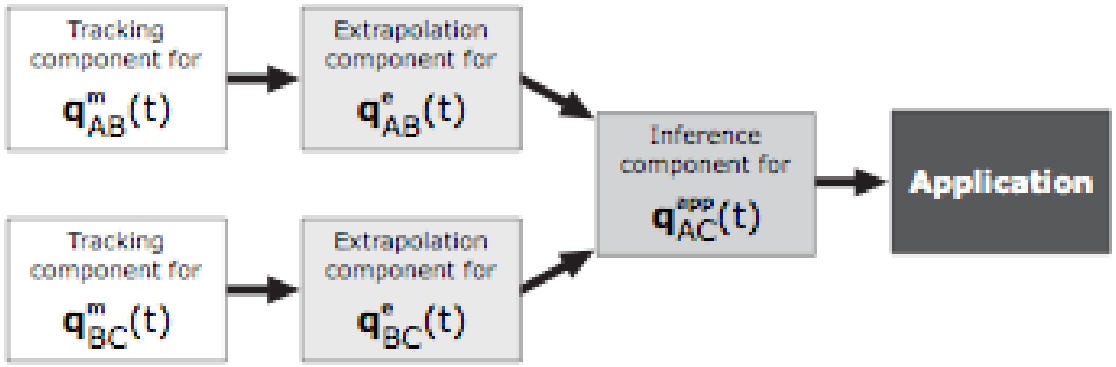


Data Flow Graph from Spatial Graph

From technical report…

- Every edge q_{AB} in spatial relationship graph must be measured or computed
- For this, we can set up data flow graph of communicating components
- Construction of data flow graph is based on attributes





Attributes

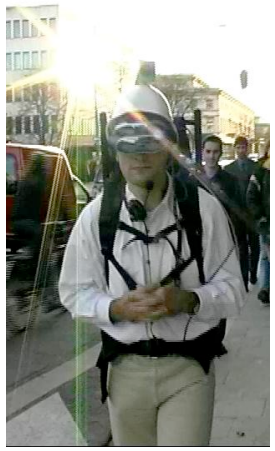
- Spatial relationships have attributes
  - E.g. latency, frequency, cost, confidence, accuracy
- Two basic assumptions about attributes:
  - Attributes change “more slowly” than measurements themselves…

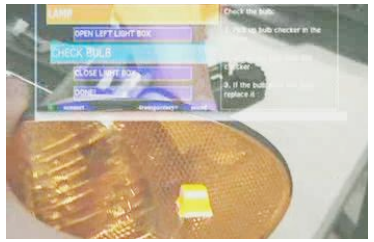
  …thus, it pays off to set up a data flow graph in the background
  - Attributes of inferred measurements can be described without actually inferring the measurements

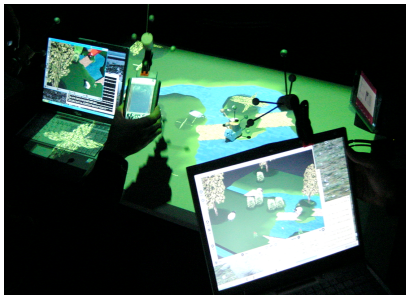
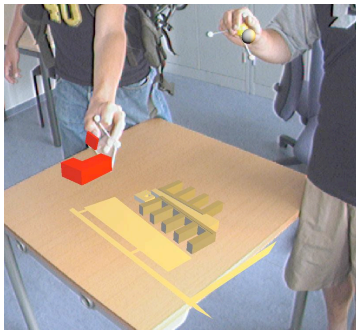
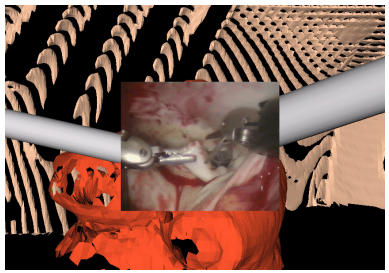
  …thus, we can compare the results of speculative data flow graphs without actual data flow

DWARF in Brief

- Framework for Mobile AR in ubiquitous computing environments
- Example scenarios:
  - Navigation (Pathfinder)
  - Maintenance (TRAMP)
  - Multi-Player Game (SHEEP)
  - Collaborative Building Design (ARCHIE)
  - Medical (HEART)

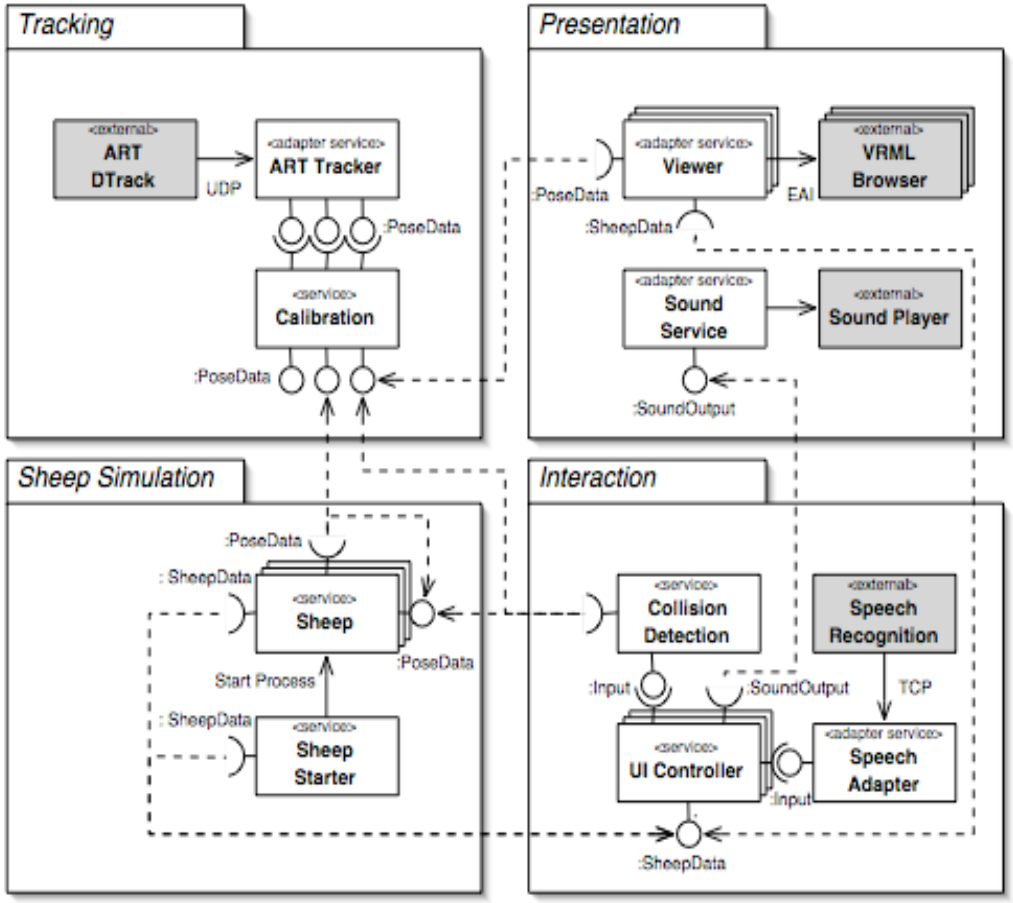




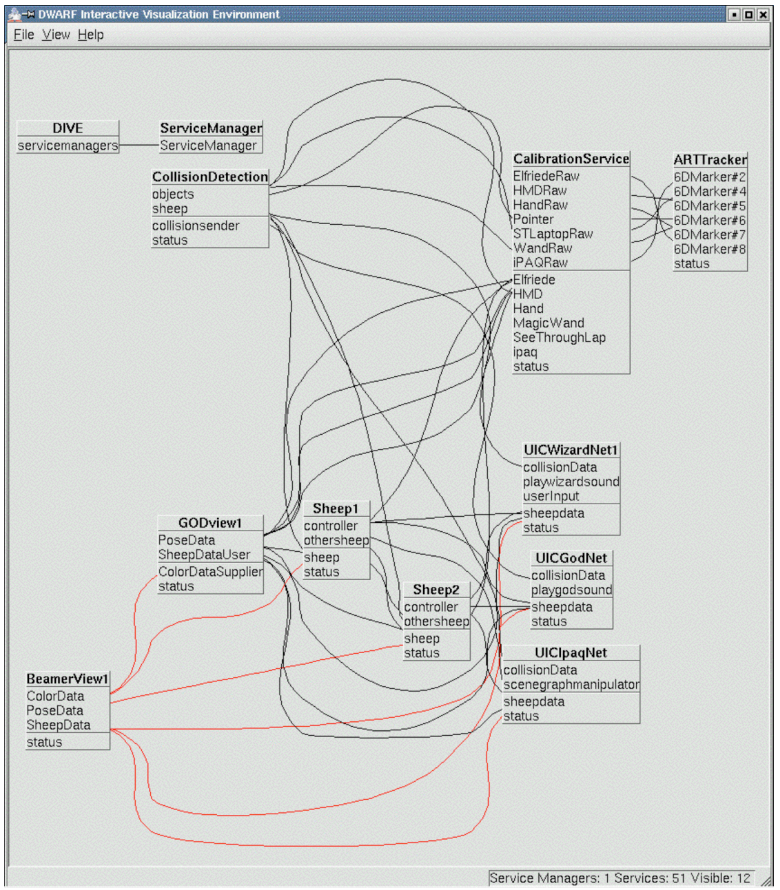





DWARF Consists of Distributed Services

During design…

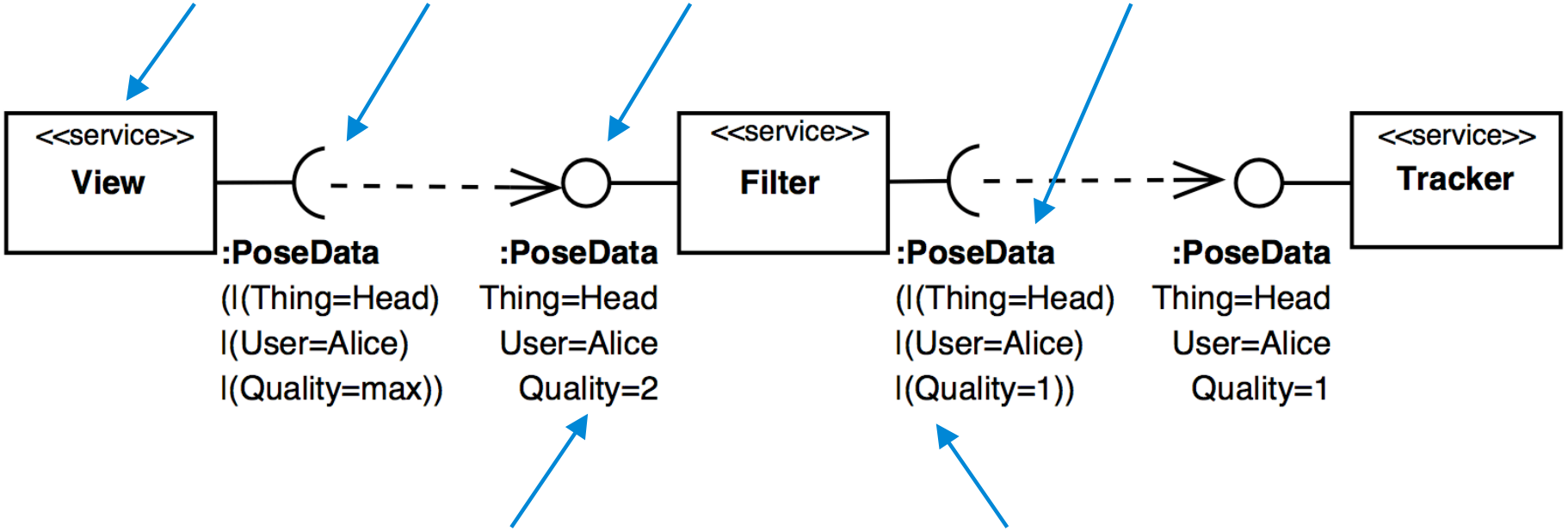


…and at run time



Services in DWARF

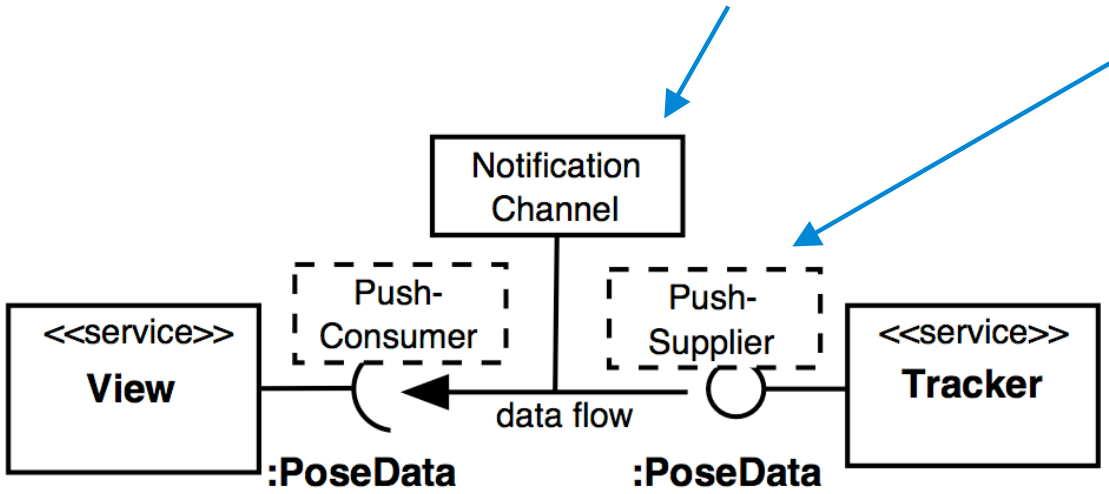
- Services have Needs and Abilities, which have types



- Abilities have Attributes, Needs have Predicates.
- These can be set at runtime.
- One service's Needs depend on other services' Abilities.
- Distributed CORBA-based Middleware establishes connections for communication between services (management, lookup, connection)

Communication mechanisms

Needs and Abilities communicate via connectors, which have protocols

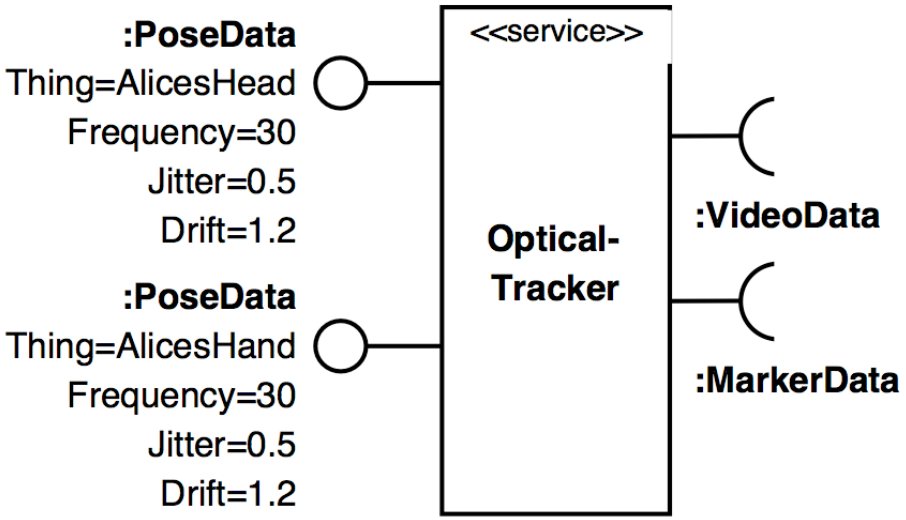


- Connectors so far:
  - CORBA structured “push” events, using strongly typed data,
    - e.g. *struct PoseData { double x,y,z; … }* in IDL
  - CORBA method calls
  - Shared memory (for local video transfer)

Mapping Ubitrack onto DWARF

- I propose a simplistic mapping:
  - Data flow components are DWARF Services
    - Trackers, filters, interpolators, extrapolators, inference components…
  - For each spatial relationship a service can compute, it has one ability of type *PoseData*
  - The relationship's attributes, and the identity of the objects related, are mapped onto the Ability's attributes
  - The communication protocol uses CORBA events or CORBA get…() method calls
  - Components like trackers have other needs, e.g. for configuration or video data

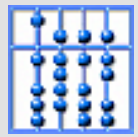
    …but that isn't relevant here



Starting Services on Demand

- Some of a service's attributes are relatively independent of the actual data it processes
- …thus, we can describe services that are not actually running…
- …and start them on demand, when they're needed

```
<service name="OpticalTracker"
  startCommand="/usr/bin/mytracker Alice.conf">

  <attribute name="Room" value="Studio"/>
  <attribute name="Lag" value="0.01"/>
  <attribute name="Jitter" value="0.5"/>
  <attribute name="Drift" value="1.2"/>

  <need name="markerData" type="MarkerData" …/>
  <need name="videoStream" type="VideoStream"…/>

  <ability name="relation1" type="PoseData">
    <attribute name="Thing" value="AlicesHead"/>
    <attribute name="RelativeTo" value="Studio"/>
    <connector protocol="NotificationPush"/>
  </ability>

  <ability name="relation2" type="PoseData">
    <attribute name="Thing" value="AlicesHand"/>
    <attribute name="RelativeTo" value="Studio"/>
    <connector protocol="NotificationPush"/>
  </ability>

</service>
```

Dynamic Attribute Changes

- So far, all attributes were specified in static XML files
- However, they can be changed at run time as well…
  - By services
    - e.g. when a tracker recognizes its accuracy is going down (optical tracker in failing light conditions)

  …Then, the middleware can select a “better” tracker for the application requesting it
  - And by the middleware
    - depending on other services found in the system
    - according to certain rules

  …that lets the middleware construct adaptive data flow graphs

Binding Attributes: Configuration

- The attributes of a service's abilities depends on how its' needs are satisfied
- For example, an optical tracker can only detect things it has marker descriptions for
- For each marker description found, the service gets a new ability
- The middleware can do this in the background, before an ability is actually requested

```
<service name="OpticalTracker"
  startCommand="/usr/bin/mytracker Alice.conf">

…
  <need name="markerData" type="MarkerData"
        predicate="(Thing=*)">
        <connector protocol="ObjrefImport"/>
  </need>

  <ability name="relation1" type="PoseData"
   isTemplate="true">
   <attribute name="Thing"
                    value="$(markerData.Thing)">
    <connector protocol="NotificationPush"/>
  </ability>

</service>
```

Binding Attributes: Cloning

- Some services can exist arbitrarily often
- For example, the Interpolation, Extrapolation, and Filtering components
- The middleware can instantiate these (“cloning”) in the background, depending on other available services

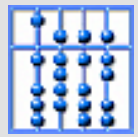
```
<service name=”MyFilter” isTemplate="true">
  startCommand="/usr/bin/myfilter”>

  <need name=”input" type=”PoseData"
   predicate=”(&(Thing=*)(RelativeTo=*)
                (Jitter>0.2))”…/>

  <ability name=”output" type="PoseData”>
   <attribute name="Thing"
                value="$(input.Thing)">
   <attribute name="RelativeTo "
                value="$(input.RelativeTo)"> …
   <attribute name=”Jitter "
                value="$(input.Jitter*0.1)"> …
  </ability>
</service>
```


Binding Attributes: Cloning (2)

- The same technique works for the Inference components, too
- The middleware can instantiate these background, depending on other available services
- Recursively, this forms chains of services

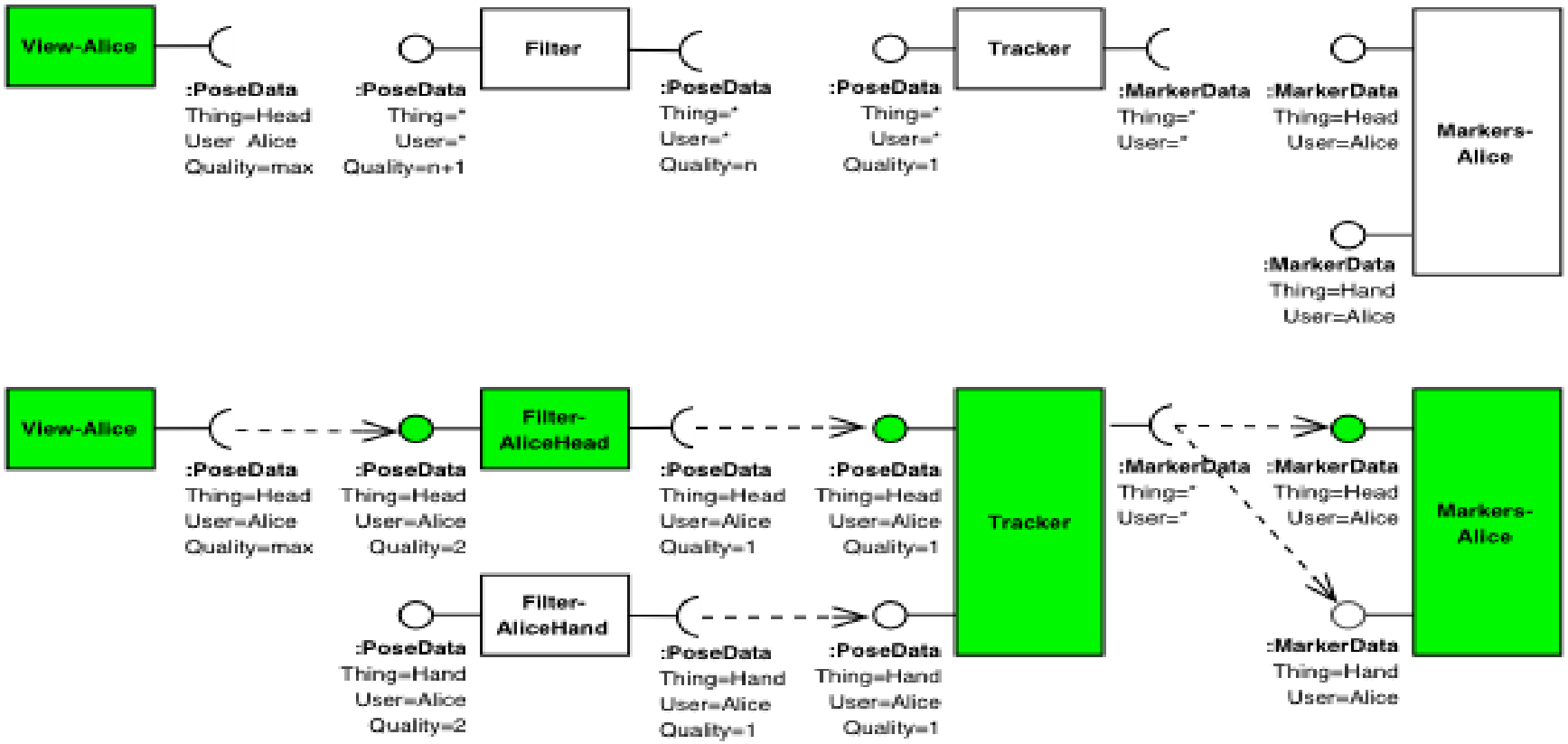
… even forests that grow exponentially

```
<service name=”MyInferrer” isTemplate="true">
  startCommand="/usr/bin/myinferrer”>

  <need name=”input1" type=”PoseData"
   predicate=”(&(Thing=*)(RelativeTo=*))”…/>

  <need name=”input2" type=”PoseData"
   predicate=”(&(Thing=*)
              (RelativeTo=$(input1.Thing)))”…/>

  <ability name=”output" type="PoseData”>
   <attribute name="Thing"
               value="$(input2.Thing)">
   <attribute name="RelativeTo "
               value="$(input1.RelativeTo)"> …
</ability>
</service>
```

Branching and Selection

- Middleware finds graphs of “potential” services in background
- When user (or application) requests a particular ability of a particular service, the appropriate chain is started up

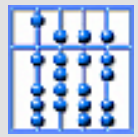


Adaption: Feedback Loop

- Feedback loop:
  - Services change their attributes according to measurements or calculations they make
  - Depending on the attributes, the middleware constructs data flow graphs
  - Depending on the data flow graphs, services are reconfigured to make different measurements or to calculate different values
- Inputs:
  - the measurements depend on the environment
  - And the data flow depends on the Needs of the application
- Such feedback loops can
  - Adapt to changing circumstances

  … but also be chaotic and unstable

  … or end up in degenerate attractors



Decentralized Algorithm

- The proposed algorithm is fairly simple to implement in a distributed fashion
  - The implementation of the DWARF middleware is based on *Service Managers*, which run on each computer in the network
- In fact, the branching of service graphs benefits from distribution
- Of course, scalability and performance may become issues
  - “Damping” rules needed to keep service graphs from exploding
  - Local middleware must react quickly to attribute changes of “relevant” services

Implementation Status

- Binding of attributes works only during cloning
  - E.g. create new filter for a certain tracker
- Evaluation of expressions not implemented yet
  - No “$(myNeed.myAttribute+1)” expressions
  - Only “Wildcard attributes”:
    - `<attribute name="Thing" value="*">`
      in Need definition; is equivalent to
    - `<attribute name="Thing" value="$(need.Thing)>”`
- Attribute changes of connected services are not propagated
- Services are not notified when their attributes change
- Service Managers do not scale well to thousands of services

What Needs to be Done

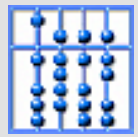
- Formalize it:
  - A template service description maps Q onto Q'
- Investigate it:
  - Find set of attributes that can be evaluated decentrally
  - Test middleware behavior with thousands of service descriptions
- Implement it:
  - Implement attribute evaluation scheme
  - Notify Services of attribute changes
  - Propagation of attribute changes between service managers
- Improve current middleware performance:
  - Better service location: beyond SLP (…)
  - Colocated communication to improve performance (…)

Service Location: Beyond SLP

- Currently, the service location mechanism uses SLP, which
  - uses broadcast/multicast queries
  - supports attributes and boolean predicates
  - is designed for fairly static services, e.g. printers
  - could span networks using federated directory agents
    - If we had an implementation of that
- What else could we use?
  - Multicast DNS: announcements, but no boolean predicate support
  - Implement some peer-to-peer resource finding algorithm using distributed indexes
  - Perhaps implement an own SLP directory agent

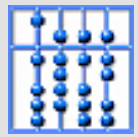
Efficient Communication: Colocation

- Currently, each DWARF service is a separate process
  - That creates communication overhead However, there is no compelling reason for that
- One process can implement multiple services and
  - register them all with registerService()
  - create new services on demand with registerServiceLoad()
  - if these service communicate using method calls, the ORB passes the call through directly
- We could support this generally by copying from COM
  - compiling C++ Services to shared libraries
  - write a loader process to load them
  - keep transparent for Services, using CorbaInit or Template Service
- However, Notification Service channels are in *notifd*, so...
  - handle 1-to-1 connections directly in loader process
  - link *libAttNotification* into loader process, too



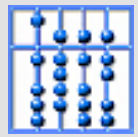
Discussion

- Strengths
  - One solution for resource discovery, configuration, adaption
  - Completely decentralized
  - Builds on existing framework
- Weaknesses
  - It may not be possible to evaluate all attributes in a piecewise, decentralized fashion
  - Distributed, “heavyweight” middleware creates overhead
  - It may not scale, in practice



Looking forward

- Where are we now?
  - We have an idea for a distributed solution
  - We have a partial implementation
- Open questions:
  - Is a decentralized attribute evaluation scheme enough?
  - Can we keep exploding search graphs under control?
  - Which attributes should we choose?
  - How should we specify the attribute dependencies?
  - Will it scale?
  - How do we integrate with OpenTracker / Studierstube?
- What should happen next?
  - Formalize, investigate, implement, optimize



Ubiquitous Tracking using the DWARF Middleware

Ubiquitous Tracking Project Workshop

Asa MacWilliams

Thank You for Your Attention!

Any Questions?

macwilli@in.tum.de