Visualizing Distributed Systems of Dynamically Cooperating Services

SEP Presentation

Daniel Pustka
pustka@in.tum.de

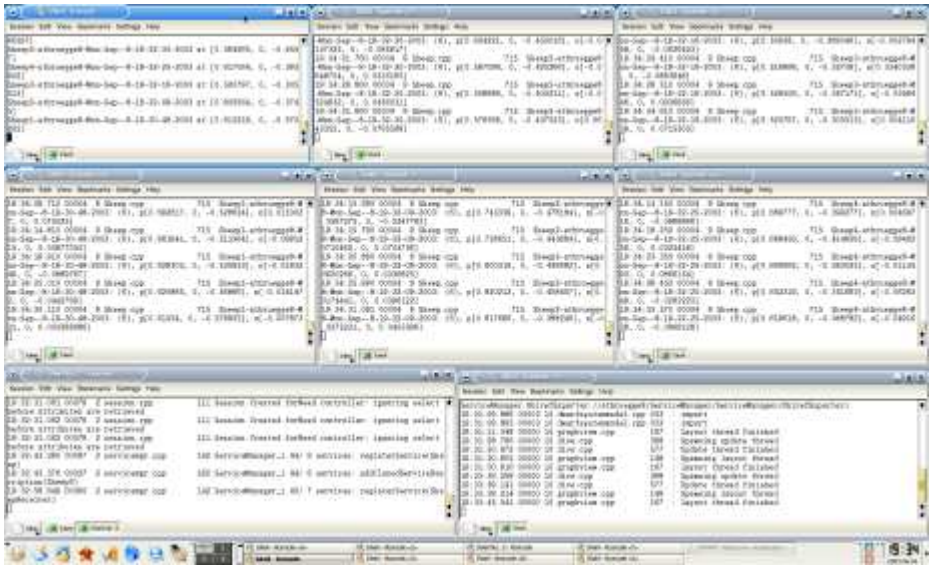
09.09.2003

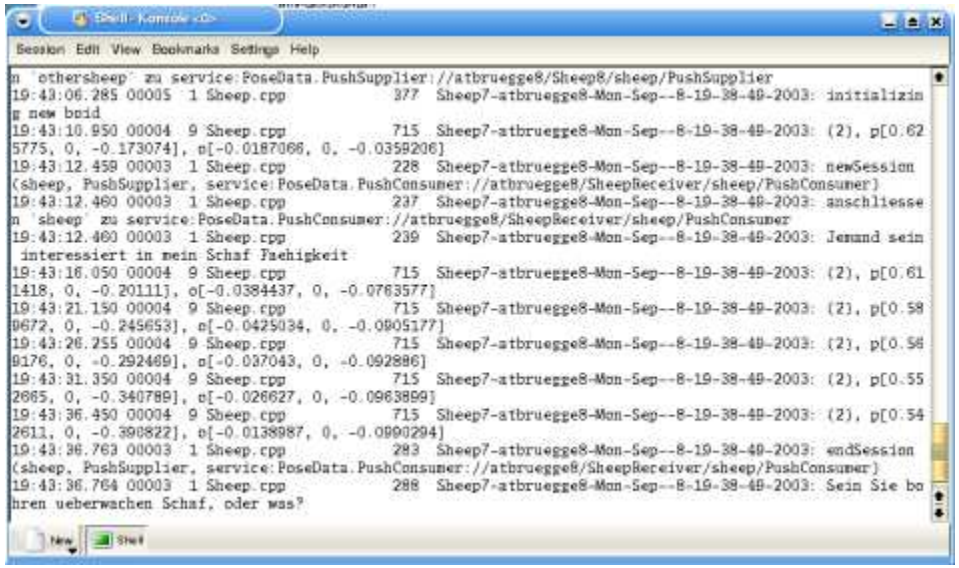
Outline

- Motivation
- Requirements
- Demo
- Implementation
  - Subsystem Decomposition
  - Graph Layout Tools
  - Debugging
- Future Extensions
- Summary

Motivation

- Finding errors in distributed systems is difficult
- Communication is an additional source of errors
- Necessary information is spread all over the network
  - DWARF: Each service manager only knows local services
- It is difficult to find relevant data in diagnostic output
- Therefore we need a visualization tool for DWARF

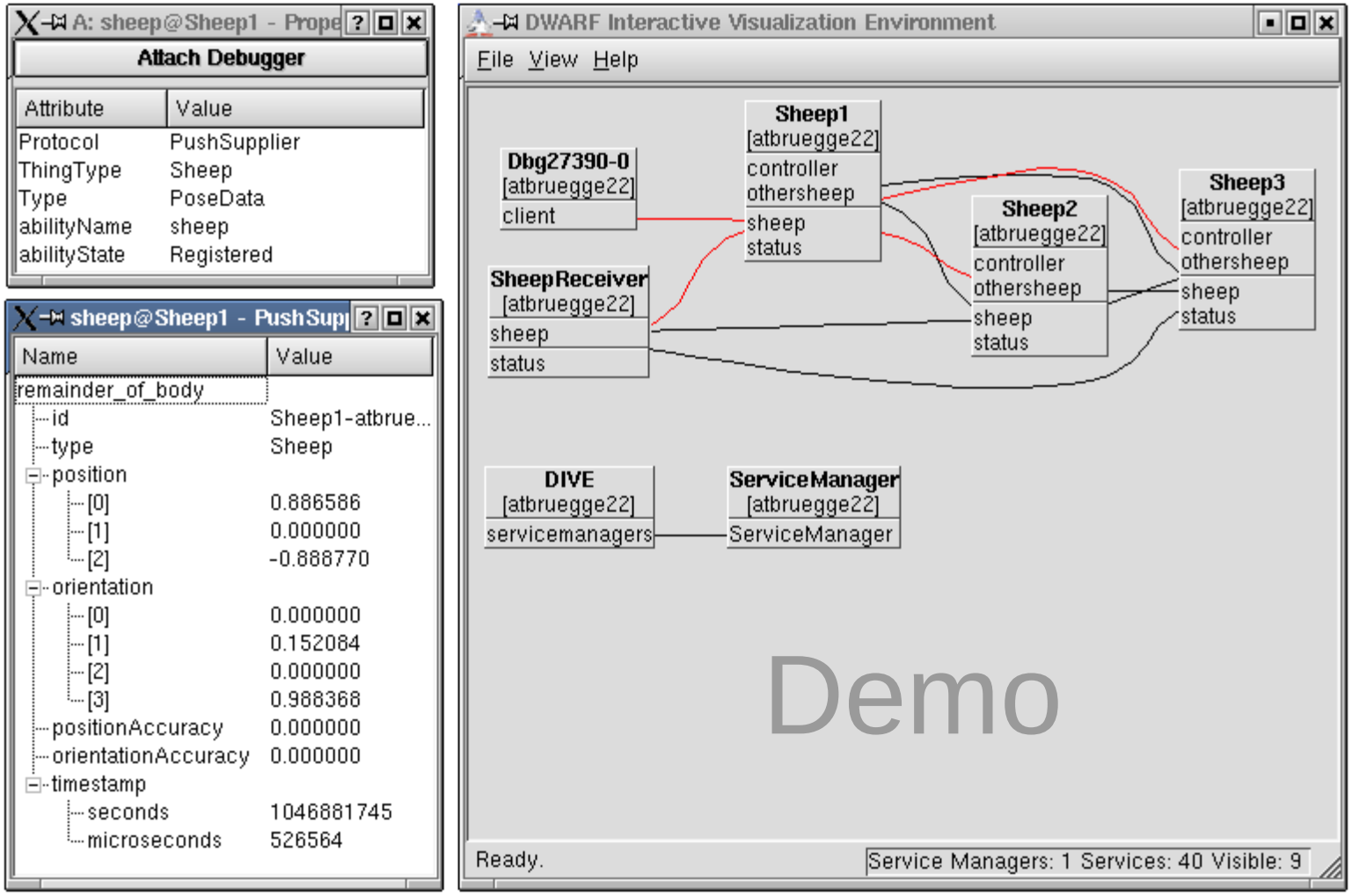




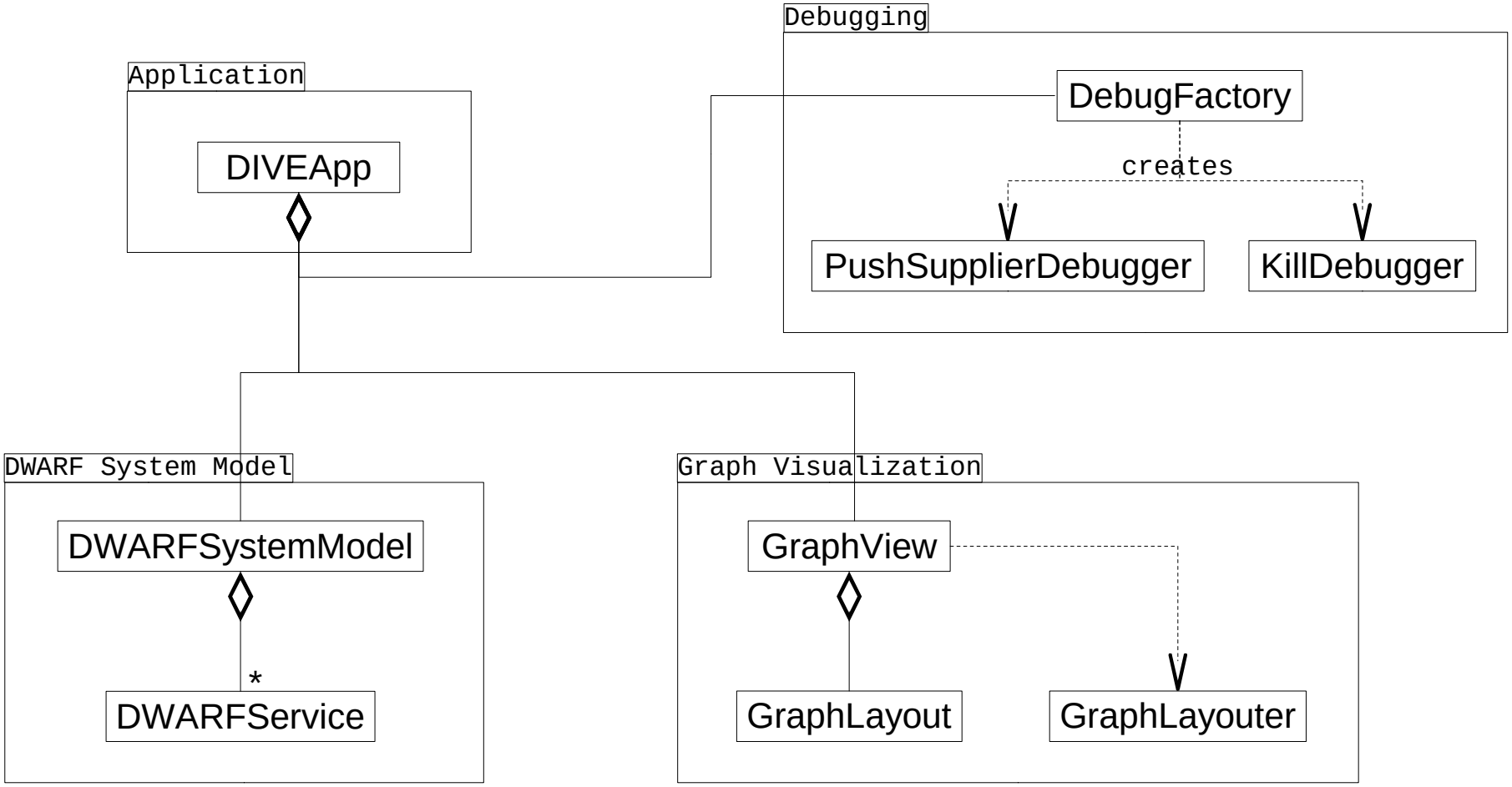
Requirements

- Collect structural information
  - Connect to all service managers
  - Combine their knowledge of the system
- Display structural information
  - Service connections as graph
  - UML class diagram style
  - Automatic graph layout
  - Service attributes
- Debugging
  - View CORBA events

The User Interface of DIVE



Subsystem Decomposition



Basic Program Flow

- Find all running service managers
  - Create a new DWARF service with a „ServiceManager“ need
- Data Aquisition
  - Connect to all known service managers
  - Read their information about services and connections
  - Store this in the DWARF System Model subsystem
- Graph Display
  - Create a graph description from the DWARF System Model
  - Pass it to the Graph Visualization subsystem
  - Call the graph layout tool
  - Display the DWARF system as a graph

Graph Layout Tools

- Fortunately many existing tools are available
- Requirements
  - Useful layout features
    - Records
    - Clusters/subgraphs
    - Incremental layout
  - Preferably open source
  - Should be under active development
  - Must be usable as a library or stdin/stdout filter
- Evaluated 12 Tools
- Final decision: Use the Dot layouter from the Graphviz collection (AT&T labs), but integrate it using the bridge pattern.

Debugging Subsystem

- Main design goal: extensibility
  - Abstract factory design pattern
  - Each debugger can provide its own Qt user interface
- Currently implemented debuggers
  - PushSupplierDebugger
    - Receives CORBA structured events from a „PushSupplier“ ability
    - Automatically decodes embedded data structures
  - KillDebugger
- Demo

Future Extensions

- More debuggers
  - CORBA method invocations
  - Image viewer
  - 3D view for PoseData
- Using DIVE for DWARF system design
  - Manipulate service descriptions
  - Remote service configuration
  - Saving DWARF applications
  - Visual programming
- Graph drawing improvements
  - Better performance through dynamic and incremental updates
  - Edge labels

Adding New Debuggers

- **Step 1:** Implement the debugger
  - Qt user interface, if necessary
  - Can use DebugService to handle connections automatically

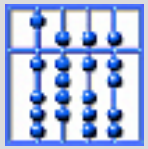
```
class PushSupplierDebugger : public QObject, public POA_DWARF::SvcProtPushConsumer {
   ...
   DebugService* m_pDebugService;

   PushSupplierDebugger( QWidget* pParent, const DWARFService&,
                         const DWARFService::Ability& );

   void push_structured_event( const CosNotification::StructuredEvent& event );
};


PushSupplierDebugger::PushSupplierDebugger( QWidget* pParent, const DWARFService& Service,
                                            const DWARFService::Ability& Ability ) {
   ...
   m_pService = new DebugService( "Dbg", "PushConsumer", Service, Ability, _this() );
   ...
}
```

- **Step 2:** Add the debugger to the DebugFactory
  - Derive from DebuggerCreator and implement HaveDebugger and CreateDebugger methods



Summary

DIVE

- works
- helps keeping an overview of dynamic distributed DWARF systems
- has successfully been used to demonstrate the DWARF middleware and applications built on top of it
- makes learning curve for DWARF less steep
- is used by almost all DWARF developers
- can be extended to become even more useful

Questions?

Thank you for your attention!