A Generic Self-Repair Approach for Overlays

Barry Porter, Geoff Coulson, and François Taïani

Computing Department, Lancaster University, Lancaster, UK
(barry.porter,geoff,francois.taiani)@comp.lancs.ac.uk

Abstract. Self-repair is a key area of functionality in overlay networks, especially as overlays become increasingly widely deployed and relied upon. Today's common practice is for each overlay to implement its own self-repair mechanism. However, apart from leading to duplication of effort, this practice inhibits choice and flexibility in selecting from among multiple self-repair mechanisms that make different deployment-specific trade-offs between dependability and overhead. In this paper, we present an approach in which overlay networks provide functional behaviour only, and rely for their self-repair on a *generic self-repair service*. In our previously-published work in this area, we have focused on the distributed algorithms encapsulated within our self-repair service. In this paper we focus instead on API and integration issues. In particular, we show how overlay implementations can interact with our generic self-repair service using a small and simple API. We concretise the discussion by illustrating the use of this API from within an implementation of the popular Chord overlay. This involves minimal changes to the implementation while considerably increasing its available range of self-repair strategies.

1 Introduction

Overlay networks are quintessential examples of decentralized distributed systems. They consist of collections of software nodes, usually one per physical host, which form a logical topology and provide a mutually desired service. Many overlays require no managed infrastructure, and are therefore suitable to be deployed dynamically and on-demand in any physical network.

Self-repair is a key area of functionality in overlay networks, especially as overlays become increasingly widely deployed and relied upon. Today's common practice is for each overlay to implement its own self-repair mechanism. For example, Chord [1] is a popular distributed hash table (DHT) overlay that employs a self-repair mechanism in which each node maintains a list of the next M nodes following it in a ring structure, and this list is continuously refreshed so that if a node's immediate clockwise neighbour fails, the node becomes linked to the next live node in the list instead. Furthermore, this list can be additionally used to redundantly store a node's application-level data in case the node fails. As another example, Overcast [2] is a tree-based content dissemination overlay that employs a self-repair mechanism which attempts to ensure that a tree node always has a 'backup parent' in case its current parent fails. This is achieved by having each node maintain a list of its ancestors, so it can choose a replacement parent from this list if its present parent fails.

But there are drawbacks to this 'per-overlay' approach to self repair. The obvious one is that it leads to duplication of effort. This is especially the case in situations where many overlay types within a 'class' (e.g. DHT overlays) ultimately use similar self-repair approaches with minor differences to suit the specifics of the particular overlay. But a more fundamental drawback is that the approach inhibits choice and flexibility in selecting from among alternative self-repair mechanisms that make different deployment-specific trade-offs between dependability and overhead. For example, where resources (e.g. free hosts) are plentiful, it may be appropriate to recover failed nodes by restoring them on other hosts. Alternatively, where resources are scarce it may be better to allow neighbouring nodes to take over the responsibilities of their failed peers.

Motivated by such considerations, we have developed an approach to self-repair in which overlay networks provide functional behaviour only, and rely for their self-repair on a *generic self-repair service*. Thanks to this separation of concerns, the application developer can be presented with two clear areas of *independent* choice: i) which overlay network to use, based on what that overlay is designed to do and how it achieves it, and ii) how that overlay should defend itself against node failure, based on the selection of an appropriate dependability/ overhead trade-off, and expressed as easy-to-understand configuration options of the generic self-repair service.

In this paper we focus on API and integration issues of this approach. In particular, we show how overlay implementations interact with the generic self-repair service using a small and simple API. We concretise the discussion by illustrating the use of this API from within an implementation of the popular Chord overlay. This is shown to involve minimal changes while considerably increasing the range of self-repair strategies available to Chord.

In the rest of this paper, we first introduce in section 2 the overall architecture of our generic self-repair service and its APIs. Then in section 3 we present the above-mentioned Chord-based case study of the use of the service's API. Finally, section 4 discusses related work, and section 5 offers conclusions.

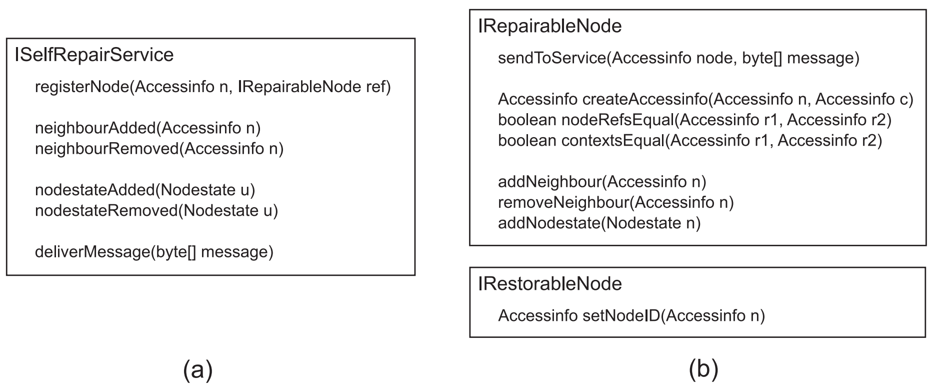
2 The Generic Self-Repair Service and its APIs

Our approach is based on the afore-mentioned separation of concerns, achieved by encapsulating all overlay self-repair concerns in a generic-but-tailorable *self-repair service*. Its design has been guided by the inherently decentralized nature of the overlays that it supports. Thus, an instance of the service runs on each applicable host (i.e. hosts that support overlay nodes that want to use the service), and the various service instances communicate in a peer-to-peer fashion to perform their respective functions.

The service comprises three distinct sub-services: i) a *distributed backup service* which takes key overlay state from a node and stores it in a 'safe' place (for example, at another overlay node) in case the node fails; ii) a *failure detection* *service* which checks neighbouring nodes for failure, and informs the recovery service when a failure occurs; and iii) *a recovery service* which uses previously backed-up data from the backup service to make appropriate repairs to the failed region of overlay. More detail on these is available in the literature [3, 4].

Between the self-repair service and the overlay there exists a two-way generic API; at one side is an API belonging to the service which permits exposition and guidance by the overlay with regard to its key state elements, and at the other side is an API allowing management of the overlay by the service.

The service does *not* therefore attempt to be fully transparent to the overlay. Rather, overlay nodes and self-repair service instances cooperate using explicit two-way interaction, whereby an overlay node calls methods on its local service instance, and a service instance calls methods on the overlay node it is supporting. As will be seen, this 'dialogue' is crucial in allowing *overlay-specific needs* to be taken into consideration. While allowing this expressiveness, the API is designed to be as simple as possible, and the relevant interfaces and methods, discussed in detail below, are shown in figure 1.



```
ISelfRepairService
    registerNode(Accessinfo n, IRepairableNode ref)

    neighbourAdded(Accessinfo n)
    neighbourRemoved(Accessinfo n)

    nodestateAdded(Nodestate u)
    nodestateRemoved(Nodestate u)

    deliverMessage(byte[] message)
```

(a)

```
IRepairableNode
    sendToService(Accessinfo node, byte[] message)

    Accessinfo createAccessinfo(Accessinfo n, Accessinfo c)
    boolean nodeRefsEqual(Accessinfo r1, Accessinfo r2)
    boolean contextsEqual(Accessinfo r1, Accessinfo r2)

    addNeighbour(Accessinfo n)
    removeNeighbour(Accessinfo n)
    addNodestate(Nodestate n)

IRestorableNode
    Accessinfo setNodeID(Accessinfo n)
```

(b)

Fig. 1. The interfaces used in our architecture: (a) The self-repair service's main interface, and (b) Interfaces to be implemented by overlay nodes

Because of the 'two-way' nature of the API, we need overlay nodes to conform to a model and a semantic that is well-understood by the service. To this end, we require that overlay implementations structure their nodes in terms of two key abstractions: *accessinfos* and *nodestates*. Based on these abstractions, which are described below, the self-repair service can *inspect* the characteristics of each node in terms of both *topology* and *state*, and can also *adapt* the topology and state as required to carry out repairs. Figure 2 depicts the full 'model' of an overlay node and its interactions with the self-repair service. The below subsections define and discuss accessinfos, nodestates and the interactions between the service and the overlay.

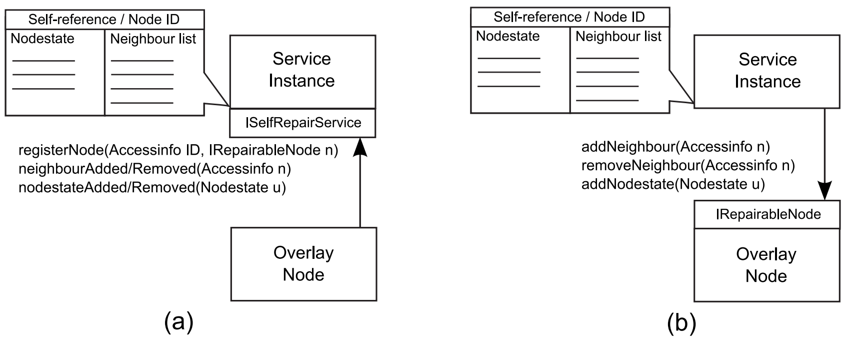

Self-reference / Node ID
Nodestate
Neighbour list
Service Instance
ISelfRepairService
registerNode(Accessinfo ID, IRepairableNode n)
neighbourAdded/Removed(Accessinfo n)
nodestateAdded/Removed(Nodestate u)
Overlay Node
(a)
Self-reference / Node ID
Nodestate
Neighbour list
Service Instance
addNeighbour(Accessinfo n)
removeNeighbour(Accessinfo n)
addNodestate(Nodestate u)
IRepairableNode
Overlay Node
(b)

Fig. 2. API interactions: (a) An overlay node exposing its accessinfos and nodestates; and (b) The self-repair service inspecting and adapting these (e.g. at repair time)

Accessinfos Accessinfos are used to expose the connectivity of the node with its 'neighbours' in the target overlay, and also to enable self-repair service instances to communicate with each other. This communication, which is useful for example to allow the service to send a backup of a node to another node, is achieved using the overlay's own topology. In terms of its representation, an accessinfo is a record that refers to an overlay node and encapsulates sufficient information to allow a message to be sent to that node. The internals of an accessinfo are entirely opaque to the self-repair service, and they are assumed to be 'serializable' so that they can be marshalled for transport and storage purposes.

When an overlay node first comes into existence, it is expected to provide its local self-repair service instance with an accessinfo that refers to itself, which is used to identify and associate various kinds of data with the node. This is achieved by calling *ISelfRepairService.registerNode(Accessinfo nodeRef, IRepairableNode n)*. This call also provides the self-repair service instance with a local object reference *n* on which the service can call overlay-side API methods.

Following this self-advertisement, each node is expected to keep its local self-repair service instance informed about changes to its 'local' topology—i.e. its connectivity to neighbouring nodes. This is achieved using *ISelfRepairService.neighbourAdded/Removed(Accessinfo n)*. With the information passed in these calls, the self-repair service instance is able to communicate with peer instances associated with the given neighbours by using *IRepairableNode.sendToService(Accessinfo n, byte[] message)*, illustrated in figure 3.

The above deals with basic topology management. However, in some cases this is not enough because it does not take into account the various topological 'roles' that might be played by certain nodes in certain overlays. For example, ring overlays may comprehend the roles of 'successor' and 'predecessor', while tree overlays may comprehend the roles of 'parent' and 'child'. To enable such semantic information to be expressed by the overlay, accessinfos can be 'tagged' by the overlay with arbitrary contextual information. As with accessinfos themselves, the nature of this information is opaque to the self-repair service.

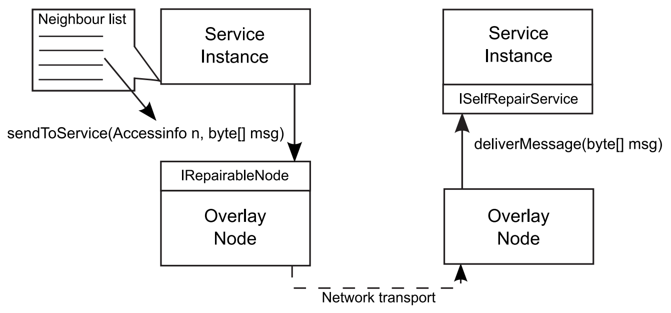


Fig. 3. Neighbours exposed by the overlay are used by the service to send data to other service instances

Nodestates Nodestates are used to encapsulate state that an overlay node is interested in having restored when the node is recovered. They are optional and do not need to be used by overlays that don't need to maintain persistent state. Like accessinfos, nodestates are assumed to be 'serializable', and the internals of nodestates are opaque to the self-repair service.

Overlay nodes pass nodestates to the self-repair service using *ISelfRepairService.nodestateAdded/Removed(Nodestate u)*. The intention is that if there is a failure, *IRepairableNode.addNodestate(Nodestate u)* can be called by the self-repair service on an appropriate target node to restore the data encapsulated in these nodestates to the overlay (the issue of which node to select for this restoration is discussed in detail in [3]).

Like the contents of nodestates, the implementation of *addNodestate()* is entirely up to the overlay. For example, a DHT overlay node may expose each file stored locally as an individual nodestate unit, and implement *addNodestate()* to map to the DHT *store()* operation, thereby routing the data to the correct place in the overlay. Alternatively, a super-peer in a Gnutella-like overlay may store its resource index as a nodestate, and implement *addNodestate()* as a 'merge' operation to merge any existing local resource index with the provided one.

Repair actions We have demonstrated above how our service can modify the topology and re-distribute the state of an overlay using a common 'model' with the general IRepairableNode interface. Beyond this base API, we support additional repair strategies and overlay input though "progressive disclosure"; a set of optional interfaces available to the overlay developer wishing to have a greater understanding of, or level of control over, the service's operations. This is exemplified here through the *IRestorableNode* interface, as shown in figure 1.

This interface is designed to allow failed overlay nodes to be fully *restored* on alternate hosts, as opposed to compensatory topology modification. This strategy is useful in Grid-like deployments, where resources may be more plentiful. Restored nodes must be provided with the 'node ID' accessinfo of the failed node they are replacing (an accessinfo originally used with *registerNode()*), achieved with the *IRestorableNode.setNodeID()* method. Both of these repair types are generically applicable to a range of overlay networks; a discussion of this, and of the mechanics of a safe, decentralized repair approach, is provided in [3].

Following this example, different 'repair strategies' can be developed and used when appropriate, according to the current state of the overlay's deployment environment, provided any additional interfaces relating to specific repair strategies are implemented. Implementing such interfaces allows the overlay to configure which strategies it would like to support, and indeed further expansion of the overlay-side API can allow arbitrary overlay guidance on how a repair strategy is performed, achieved by the strategy querying the overlay before repair enactment. A discussion and evaluation of the benefits of dynamically selecting different repair strategies at the time of repair is available in [4].

3 Case study

We now provide a detailed view of the overlay developer's task in making an overlay compatible with our self-repair service. We use Chord [1] as an example, as it is well-known and easy to understand without being trivial, though we have also modified TBCP [5] (an application-level multicast overlay) in a similar way. The presentation here is based on an actual modification of an existing Chord prototype [6] as used in our GridKit middleware. We assume some familiarity with Chord, as we do not have space to discuss it in detail here.

3.1 Instantiating the abstractions

Chord has two main topological characteristics: (i) a ring structure, in which nodes are linked clockwise and anti-clockwise by 'successor' and 'predecessor' links respectively; and (ii) the use of per-node 'finger tables' that provide O(log N) routing for any key (as the ring alone would only provide O(N) routing).

For the purposes of this case study we decided to expose the ring structure in terms of accessinfos, but to model finger tables in terms of nodestates. This latter choice makes sense because we know that the finger table will be refreshed shortly after a repair anyway, and so use the finger table data only to speed up this process. However, it would also have been possible to expose finger tables in terms of accessinfos by calling *neighbourAdded/Removed()* as each finger link changes, with an appropriate 'finger' context (see below).

Having made this decision, the next step is to define an accessinfo class as shown in figure 4. The Chord implementation we modified uses Java RMI as its communication protocol, and so the *RemoteNode* reference is a wrapper around an RMI remote reference (and Chord node ID). Our accessinfo class also has a comparator method and a 'context' variable, which is used to tag the record with contextual data of 'successor', 'predecessor' or 'NodeID'. It is serializable for network transport and flat storage.

We next create a 'Chord nodestate' class to contain a node's finger table (simply an array of remote references). This completes our definition of Chord

```
class ChordAccessinfo implements java.io.Serializable {
  public static final int NODE_ID = 0, SUCCESSOR = 1, PREDECESSOR = 2;
  public RemoteNode nodeReference;
  public int context;

  public ChordAccessinfo(RemoteNode nodeReference, int context) { .... }

  public boolean equals(Object o) {
    if (o instanceof ChordAccessinfo) {
      ChordAccessinfo chk = (ChordAccessinfo) o;
      if ((chk.nodeReference.equals(nodeReference))
          && (chk.context == context))
        return true;
        else
        return false;
      }
      else return false;
    }
  }
```

Fig. 4. Our `accessinfo` implementation for Chord

under the self-repair service's model. This is clearly both Chord-specific (with customized contexts), and implementation-specific (an RMI remote reference is used as our accessor detail). But again, we stress that our service does not have (or need) access to the contents of the above classes.

3.2 Using the API

To be repairable by our service, the Chord node implementation must implement seven methods, shown in figure 5. Our Java service implementation uses Java Objects to represent accessinfos and notestates, but to simplify the presentation here we have assumed 'implicit casting' to the ChordAccessinfo class (and corresponding nodestate class 'StoredFingerTable') in our code extracts.

Note the *receiveServiceMessage()* call, shown in figure 5, is a remote method call, from which the recipient overlay node calls *deliverMessage()* on its self-repair service instance to deliver the sent message (as in figure 3 in section 2).

In order to expose the overlay's topology and state to the service, the overlay node implementation also needs to call six methods on the self-repair service at various points during its execution. We call *registerNode()* on startup, then provide topology information as it becomes available or changes using *neighbourAdded/Removed()*. Successor changes occur within Chord's *stabilize()* method, of which we show an extract in figure 6.

Chord's *notify()* method was also modified in the same way to expose changes in the node's predecessor. In both cases, we decided not to internally store neighbour links as instances of our new ChordAccessinfo class, but instead to create them only when Chord interacts with our service. This decision was made because we were modifying an existing overlay, and therefore wished to make as few changes to it as possible.

Finally, when a Chord node updates its finger table, we again perform a similar procedure to that shown in figure 6, but this time using the methods

```
public void sendToService(ChordAccessinfo toNodeID, byte[] message) {
  ((ChordRef) toNode).nodeReference.receiveServiceMessage(message);
  }
public ChordAccessinfo createAccessinfo(ChordAccessinfo toNode, ChordAccessinfo useContext) {
  return new ChordAccessinfo(toNode.nodeReference, useContext.context);
  }
public boolean nodeRefsEqual(ChordAccessinfo r1, ChordAccessinfo r2) {
  return r1.nodeReference.equals(r2.nodeReference);
  }
public boolean contextsEqual(ChordAccessinfo r1, ChordAccessinfo r2) {
  return r1.context == r2.context;
  }
public void addNodestate(StoredFingerTable unit) {
  refreshFingerTable(unit.table);
  }
public void addNeighbour(ChordAccessinfo neighbour) {
  //check what kind of neighbour it is
  if (neighbour.context == ChordAccessinfo.SUCCESSOR)
    setSuccessor(neighbour.nodeReference);
    else if (neighbour.context == ChordAccessinfo.PREDECESSOR)
    setPredecessor(neighbour.nodeReference);
  }
public void removeNeighbour(ChordAccessinfo neighbour) {
  //check what kind of neighbour it is
  if (neighbour.context == ChordAccessinfo.SUCCESSOR)
    setSuccessor(null);
   else if (neighbour.context == ChordAccessinfo.PREDECESSOR)
    setPredecessor(null);
  }
```

Fig. 5. Our implementation of the IRepairableNode interface for Chord

```
private void stabilize() {
  ...

  // Updating successor
  RemoteNode successor =  getSuccessor();
  RemoteNode rn = successor.getPredecessor();
  // Check to see if the successor's predecessor is greater than Node ID
  if ((rn != null) && (ChordNodeID.inRange(rn.nodeID, myNodeID, successor.nodeID))) {
    // If it is, set node successor to the successor's predecessor
    dependabilityService.neighbourRemoved(new ChordAccessinfo(successor, ChordAccessinfo.SUCCESSOR));
    setSuccessor(rn);
    dependabilityService.neighbourAdded(new ChordAccessinfo(rn, ChordAccessinfo.SUCCESSOR));
    }
  ...
  }
```

Fig. 6. Our modified implementation of the stabilize method in Chord

```
public ChordAccessinfo setNodeID(ChordAccessinfo nodeID) {
  setChordNodeID(nodeID.nodeReference.chordID);
  return new ChordAccessinfo(myRemoteReference, ChordAccessinfo.NODE_ID);
  }
```

Fig. 7. Our implementation of *setNodeID()* for Chord

nodestateAdded/Removed(), where we remove the old finger table from persistent storage with the service, and add the new one, as nodestate.

3.3 Optional additional implementation

To allow the recovery service to choose at runtime between topology-modifying and node-restoring repair strategies, we implement the optional interface IRestorableNode, with its *setNodeID()* method as in figure 7.

We now have a version of Chord which can be maintained by our self-repair service, so that the service provides all aspects of redundancy, failure detection, and recovery. We believe that all overlay-side instrumentation is trivial for the overlay developer, as demonstrated by the code extracts above—in total, our modified version of Chord is 80 lines (10%) longer than the 'standard' version. We have not altered Chord's functional behaviour in any way, so it is fully compatible with existing applications.

Nevertheless, it is additionally possible for such applications, again with minor modifications, to themselves take advantage of the self-repair service to ensure that their data is safe across node failures. To achieve this, applications simply need to wrap their data as nodestates and call *addNodestate()* and *removeNodestate()* on the service as data is added to/removed from the local node.

4 Related work

The notion of "progressive disclosure" has similarities with the "scope control" property advocated for reflective operating systems by Kiczales and Lamping in [7], though we also use it to help configure how the recovery service works.

The 'model' of an overlay as defined by our self-repair service can be seen as a kind of 'reflection' of an overlay, i.e. a causally-connected meta model of a running system. Previous work [8] has addressed 'domain-specific' reflection of a system for fault tolerance, but our model has much closer ties to the specific *application* (i.e. overlays), rather than general distributed systems.

Our approach could also be viewed as inserting 'probes' and 'actuators' into an overlay to monitor and modify it, in a similar way to an autonomic control loop [9], with our service as the decision-making module.

There has been some other interesting work on making overlay networks easier to develop, in a similar way to that in which we allow developers to ignore self-repair concerns. iOverlay [10] and MACEDON [11] both suggest frameworks in which to develop overlays; in the case of iOverlay, the developer expresses only the 'business logic' of the overlay, and communication and other concerns are handled by the framework. MACEDON aims to make overlay development easier by using a specialized language with which to design and evaluate overlays.

While both of these efforts may be valuable in the design process, we have deliberately aimed at an API which is very close to the way in which overlays operate today, which permits both minimal effort in modifying existing implementations, and a low 'learning curve' for new implementations, as we use standard object-oriented principles.

5 Conclusion

In this paper we have described how a generic repair approach can be integrated with overlay code in a very practical way. Although we had only space to discuss one concrete example, we hope the reader can see how the approach would apply equally to other overlays. The basic approach is to define some core abstractions (accessinfos and nodestates) and an API based on allowance for application-specific expressiveness and progressive disclosure. This potentially simplifies the implementation of new overlays, which no longer need concern themselves with failure and self-repair issues, and allows many of the wide range of overlay types that exist today to benefit from our self-repair service, without losing overlay-specific semantics. This integration opens up a much wider range of repair strategies than are typically available to off-the-shelf overlays.

By providing such a separation of *dependability* and *functionality*, two concerns often particularly closely tied in overlay networks, we enable design choice based on overlay functionality, and independent dependability property specification *as appropriate to the deployment* through a standard service.

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, ACM Press (2001) 149–160
2. Jannotti, J., et al.: Overcast: Reliable multicasting with an overlay network. In: Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI). (2000) 197–212
3. Porter, B., Taïani, F., Coulson, G.: Generalized repair for overlay networks. In: Proceedings of the Symposium on Reliable Distributed Systems (SRDS), Leeds, UK (2006)
4. Porter, B., Coulson, G., Hughes, D.: Intelligent dependability services for overlay networks. In: Proceedings of Distributed Applications and Interoperable Systems 2006 (DAIS'06). Volume 4025 of LNCS., Bologna, Italy (2006) 199–212
5. Mathy, L., Canonico, R., Hutchison, D.: An overlay tree building control protocol. Lecture Notes in Computer Science **2233** (2001) 76
6. https://sourceforge.net/projects/gridkit/: (Gridkit middleware public release)
7. Kiczales, G., Lamping, J.: Operating systems: Why object-oriented? In Hutchinson, L.F.C., Norman, eds.: the Third International Workshop on Object-Orientation in Operating Systems, Asheville, North Carolina (1993) 25–30
8. Killijian, M., Fabre, J., Ruiz-García, J., Shiba, S.: A metaobject protocol for fault-tolerant CORBA applications. In: 17th IEEE Symposium on Reliable Distributed Systems (SRDS-17), West Lafayette (USA) (1998) 127–134
9. Ganek, A., Corbi, T.: The dawning of the autonomic computing era. IBM Systems Journal **42:1** (2003) 5–19
10. Li, B., Guo, J., Wang, M.: iOverlay: A lightweight middleware infrastructure for overlay application implementations. In: Proceedings of IFIP/ACM/USENIX Middleware, Toronto, Canada (2004)

11. Rodriguez, A., et al.: Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In: Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, USA (2004)