Chedar: Peer-to-Peer Middleware

Annemari Auvinen, Mikko Vapa, Matthieu Weber, Niko Kotilainen and Jarkko Vuori

Department of Mathematical Information Technology
University of Jyväskylä
P.O.Box 35 (Agora), 40014 University of Jyväskylä, Finland
{annauvi, mikvapa, mweber, npkotila, jarkko.vuori}@jyu.fi

Abstract

In this paper we present a new peer-to-peer (P2P) middleware called CHEap Distributed ARchitecture (Chedar). Chedar is totally decentralized and can be used as a basis for P2P applications. Chedar tries to continuously optimize its overlay network topology for maximum performance. Currently Chedar combines four different topology management algorithms and provides functionality to monitor how the peer-to-peer network is self-organizing. It also contains basic search algorithms for P2P resource discovery. Chedar has been used for building a data fusion prototype and a P2PDisCo distributed computing application, which provides an interface for distributing the computation of Java applications. To allow Chedar to be used in mobile devices, the Mobile Chedar middleware has also been developed.

1 Introduction

Peer-to-peer technologies have received a lot of publicity lately mainly because of Napster and other peer-to-peer systems mostly developed for distributing music and movies in the Internet. A peer-to-peer network is also well suited for sharing other resources than files, for example CPU time and storage space. Every node in a P2P network may provide resources to other nodes and consume resources the other nodes are providing, i.e. a node may serve both as a server and a client. Therefore there is no need for a central server which might become the bottleneck of the network or which failure will paralyze the whole network. Also the data traffic is more evenly distributed in the P2P network than in the centralized networks where central node's data traffic might be very large.

Gnutella [14], published in 2000, is a decentralized pure peer-to-peer protocol [15]. Gnutella servents use TCP connections for communication and the Breadth First Search (BFS) algorithm for searching resources. When a node wants to join the Gnutella network it must first find one node in the network to which to establish a connection. That node can be found for example from a web page containing a list of nodes. In Gnutella, a node usually has some pre-defined amount of connections. To find new neighbors a Gnutella node uses ping messages. A ping message is broadcasted in the network and nodes reply to it with pong messages. The node stores information about the active connections it has so it can try to connect to those when joining the network after disconnecting.

In Gnutella the search queries are broadcasted in the network. The querier sends the query message to its neighbors, which forward the query to their neighbors except the node from where the query arrived. The amount of hops the query travels can be limited by setting a time-to-live (TTL) value. Every time a node forwards the query, it decrements the value of the TTL by one. When the value of the TTL becomes zero the node drops the message. If a node owns the queried resource it sends a reply to the querier using the same route as the query came from.

Chedar differs from Gnutella in some ways. Chedar is a middleware, i.e. it offers an API for P2P applications. It contains new kinds of topology management algorithms by which the overlay topology on top of the physical network is self-organized. Those algorithms use only the local information the nodes have on their neighbors. The purpose of the algorithms is to create a network which is scalable and fault-tolerant. Chedar also has four other search algorithms implemented, in addition to the BFS that Gnutella uses. Chedar also

This work was supported in part by the Agora Center InBCT project.

1-4244-0054-6/06/$20.00

guarantees that a resource reply message can be forwarded to the querier if it still exists in the network. Chedar uses an XML-based, structured resource description and the XPath language for matching the query keywords with its resources.

This paper is organized as follows. We describe Chedar in Section 2 and its structure in Section 3. The messages Chedar uses are described in Section 4. The algorithms used for managing the topology of the Chedar network are presented in Section 5 and the paper is concluded in Section 6.

2 Chedar

We have developed the Chedar system for resource sharing and distribution. Chedar is a pure peer-to-peer middleware implemented using the Java programming language. Any application which uses the API and implements the callback functions required by the API may use Chedar and run on it. Peers, i.e. nodes in the network, communicate directly with each other using TCP connections. Chedar is developed to work in a dynamic environment where the nodes may join or leave the network whenever they want without causing significant problems to the applications running on top of Chedar. Because there are no central points in the network, Chedar is fault-tolerant and scalable. In case of link failures the topology management algorithms ensure that new peers will be contacted and the network stays connected.

Chedar can be used to distribute different kinds of resources to other nodes in the Chedar network. Distributed resources can be for example files, CPU time or storage space. Every node stores information about the resources it provides in XML format. When the node receives a query about some resource it checks by using the XPath expression whether it owns the resource.

In Chedar a neighbor's goodness is defined based on the resource replies the node receives from the neighbor to the requests the node has sent. The more the neighbor offers requested resources, the more important it is for the node. The amount of replies the neighbor has relayed to the node also affects it's goodness value. The overlay topology and the traffic in the Chedar network is managed by the Overtaking and Overload Estimation algorithms which use neighbor's goodness value as a measure for selecting which of the connections should be dropped and where to connect. Also Chedar always tries to route the resource reply to the initiator of the request. In Chedar it is possible to use multiple search algorithms, unlike in Gnutella which only uses the broadcasting search algorithm.

In our research project [4] Chedar has been used for distributed computing [9] and data fusion [13] and extended also to mobile devices [10]. Peer-to-Peer Distributed Computing (P2PDisCo) software was built on top of Chedar to speed up the training of neural networks with evolutionary computing. In the Decentralized Data Fusion System (DDFS) application each sensor node is one Chedar node. DDFS can be used to track targets based on the sensor measurement of their coordinates. Mobile Chedar is an extension to the Chedar peer-to-peer network for mobile peer-to-peer applications and has been implemented using Java 2 Micro Edition. We have also developed P2PStudio [8] monitoring application for the Chedar network to study the performance of search algorithms and the self-organizing behavior of the topology management algorithms.

3 Structure of Chedar

Chedar consists of five main components which are Connections, ConnectionManager, PropagationEngine, TopologyManager and ChedarClient. These components are illustrated in Figure 1.

3.1 Connections

The Connections include local information used by the topology management algorithms about the node's neighbors. Each neighbor is one connection object. Chedar keeps information about active connections and history data about the earlier connections in XML trees. Searches can be made to the XML tree using an XPath expression. History data also contains information about the nodes which the peer has found out from its neighbors. The nodes save the IP addresses and the TCP ports of the neighbors, the types of resources those provide and hit values per provided resource types. Hit values are described later in the next paragraph. Chedar saves also the time when the connection last replied, when the connection request has been sent to the connection and whether the request

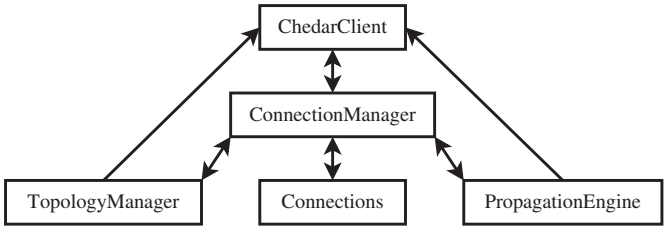

Figure 1. Main components of Chedar.

succeeded or not. Relayed hits and the number of the connection's neighbors is stored about active connections.

Every connection has three types of hit values in Chedar. First one, called *hit value*, is increased by one every time the node gets a resource reply from its neighbor. Second one is called *actual hits* and is increased when the node uses a resource the neighbor provides. *Relayed hits* values of the connection include the neighbor's neighbor nodes and the amount of the reply messages those have sent to the node through the neighbor.

3.2 ConnectionManager

The ConnectionManager manages the active connections and the history data by adding connections or removing connections according to the TopologyManager's requests. The ConnectionManager keeps a cache about the information of forwarded messages and handles all arriving messages and passes them to the classes which have informed wanting that type of messages. The ConnectionManager has also a traffic meter which measures the size of the resource messages going through the node in a given time period. The traffic meter is used by the Overload Estimation Algorithm.

3.3 TopologyManager

The TopologyManager selects the connections to establish or remove and the nodes to overtake by using the algorithms described in the Section 5. It handles *ConnectionRequest* and *ConnectionReply* messages, *NeighborListRequest* and *NeighborListReply* messages and *ServiceListRequest* and *ServiceListReply* messages which are described in the Section 4.

3.4 PropagationEngine

The PropagationEngine handles the resource messages. The application running on top of Chedar can select any search algorithm that is implemented in Chedar. Currently five different resource discovery algorithms have been implemented: BFS [12], Random Walk [11], Highest Degree Search [1, 6] and NeuroSearch [16]. Some of the algorithms are reviewed in [16]. The PropagationEngine passes the received resource request or reply message to the search algorithm specified in the message. The algorithm makes the decision where to forward the message and creates a reply message when needed. The algorithm returns to the PropagationEngine the forwarded message and a list of connections where to forward the message.

3.5 ChedarClient

The ChedarClient works as an API for Chedar. The ChedarClient provides methods for setting and getting the values of the parameters used in the algorithms. Resources can be set with the ChedarClient and it propagates events about received and sent messages and overtakings to the application. The ChedarClient also provides methods for establishing a connection to a certain node and for creating a resource request. Table 1 shows the methods that are accessible in the ChedarClient for the applications running on top of Chedar.

By implementing Chedar's monitoring interface, the iMonitor, the application may get the following events: *resourceQuerySentEvent, messageForwardedEvent, messageDiscardedEvent, resourceReplySentEvent, resourceReplyReceivedEvent, overtakingEvent, connectionRequestedEvent* and *connectionStatusEvent*.

4 Messages

There are three types of messages used for the topology management in Chedar. When a node wants to establish a new connection to another node it sends a *ConnectionRequest* message to it. The requested node sends back a *ConnectionReply* message which includes the information whether it accepts the request or not.

With a *NeighborListRequest* message the node can query a connection's neighbors, i.e neighbor's neighbors. The sender puts its own neighbors' IDs into the message. The node replies to the request by sending back a *NeighborListReply* message which includes the neighbors' IDs. The nodes save the neighbors' IDs to the history data.

A node can query the resource types its neighbor provides, e.g. file or computing time, by sending to a connection a *ServiceListRequest* message which is replied by a *ServiceListReply* message. The request message includes the resource types the sender provides and the reply message includes the replier's resource types.

Resources are searched with a *ResourceRequest* message. An application needing a resource starts a query using a resource discovery algorithm it wants. When the message arrives to a node which has the requested resource, it handles the message according to the algorithm and sends the reply message back to the initiator of the request message using the following method.

Chedar tries to guarantee that the *ResourceReply* message is forwarded to the initiator of the request message by using a simple method. Every node keeps information about *ResourceRequest* messages it has forwarded. The nodes save the ID of the message and the

startChedar()	Starts a Chedar node.
startConnecting()	The node starts establishing connections.
setMonitor(iMonitor monitor)	Sets a monitoring application for events.
pingMessage()	Checks if a Chedar node is still alive.
connect(String password)	Connects to the node. Returns true if the password is correct.
getMyID()	Returns node's ID.
getNeighbors()	Returns neighbors' IDs.
setResources(Node resource)	Sets a resource node to the XML tree.
unsetResource(String xpath)	Removes corresponding resource from the XML tree using an XPath expression.
listResources()	Returns a list of resources the node has.
createResourceQuery(String query, String algorithm, int ttl)	Creates a resource request message where the query is the searched resource as XPath, the algorithm is the used search algorithm and the ttl is a time-to-live value. Returns the id of the created message.
stopMessage(String id)	Stops forwarding the message with a given id.
setTrafficLimit(int limit)	Sets a value for the traffic limit.
getTrafficLimit()	Returns the value of the traffic limit.
getTrafficMeter()	Returns the value of the traffic meter.
resetTrafficMeter()	Sets the value of the traffic meter to zero.
forceConnection(String id)	Establishes a connection to the node specified with the id.
forceDisconnection(String id)	Disconnects the neighbor specified with id.
closeAllConnections()	Disconnects all node's connections.

Table 1. Methods accessible for applications running on top of Chedar.

IDs of the two previous nodes of the path the message arrived from. The reply message is forwarded to the initiator of the request message using the same path as the request came from, i.e. the reply message is sent to the connection where the request arrived from. This is a common way to route the replies in P2P networks because it needs only information about the previous node. In Chedar there is also other ways if the forwarding fails. If the connection to the neighbor is not available anymore, for example the neighbor has left the network, the node tries to establish a new connection to the second next node on the return path, i.e. the second previous node on the query path and sends the message there. If this does not work either, finally the node tries to establish a new connection to the initiator of the query and sends the reply message directly to it. Establishing always a direct connection to the initiator of the query would require establishing a new TCP connection, which is not always possible, e.g. in the presence of firewalls. Also keeping statistics of which nodes have relayed replies would not be possible.

5 Topology Management Algorithms

Chedar contains four algorithms for managing the topology: Node Selection for adding neighbors, Node Removal for removing neighbors, Overload Estimation for limiting the node's traffic and Overtaking for moving in the network. The algorithms have been further developed and tested in the P2PRealm simulator [7] and the behavior of the algorithms is analysed in [3].

5.1 Node Selection Algorithm

The initial list of neighbors can be obtained manually by out-of-band methods or automatically by using advertisement systems [17] or centralized entry point directories [5]. This has not been implemented in Chedar, but instead it has been left to the application running on top of Chedar. The Node Selection Algorithm handles only the case when a Chedar node already knows some nodes in the network.

When the node joins the network again it tries to establish the connections it had before leaving the network, i.e. connections saved in the active connections. In the best case it manages to establish all connections it had earlier. If the node does not manage to establish any of those connections or it needs a new connection for other reasons, it searches the next one from the history as shown in Algorithm 5.1.

First it searches connections which have hit values and tries to create a connection to one of those.

Because the node does not want to create a connection to the same node it has just dropped it searches only the connections which have not been requested in a given time. If the node did not succeed in establishing a new connection, it next searches connections based only on the time of the last request, i.e. the node has not tried to create a connection to those in a given time or at all (lacking requested information). If the node still did not successfully create a connection, it searches connections without information for hit values or request time. If the node does not have neighbors then the last way to search for a new connection is to try the connections in the history which have hit values. Then the node may select again a neighbor which it has just dropped.

Algorithm 5.1 (NodeSelectionAlgorithm)
Input: Connections h_is in node's history $H = \{h_1, ..., h_m\}$, *time* sets a limit for the time which older the previous connection request must be and *neighbors* is the number of node's neighbors.
Output: Establishes a new connection
hitsNeeded = true
timeNeeded = true
C = SearchConnections(*hitsNeeded, timeNeeded, time, H*)
for i=1 **to** $\mid C \mid$ **do**
  if EstablishConnection(c_i) **then do**
    return c_i
  end if
end for
hitsNeeded = false
timeNeeded = true
C = SearchConnections(*hitsNeeded, timeNeeded, time, H*)
for i=1 **to** $\mid C \mid$ **do**
  if EstablishConnection(c_i) **then do**
    return c_i
  end if
end for
hitsNeeded = false
timeNeeded = false
C = SearchConnections(*hitsNeeded, timeNeeded, time, H*)
for i=1 **to** $\mid C \mid$ **do**
  if EstablishConnection(c_i) **then do**
    return c_i
  end if
end for
if *neighbors* == 0 **then do**
  hitsNeeded = true
  timeNeeded = false
  C = SearchConnections(*hitsNeeded, timeNeeded, time, H*)
  for i=1 **to** $\mid C \mid$ **do**
    if EstablishConnection(c_i) **then do**
      return c_i
    end if
  end for
end if

The function SearchConnections(*hitsNeeded, timeNeeded, time, H*) returns those connections $C = \{c_1, ..., c_n\} \subseteq H$ which meet the criteria defined in the parameters. If the value of the parameter *hitsNeeded* is true, then the function only returns the connections which have hit values. If *hitsNeeded* is false the function returns the connections which do not have hit values. If the value of the parameter *timeNeeded* is true, then the function only returns the connections which have not been requested in the time defined in the parameter *time*. The function EstablishConnection returns true, if the connection was established successfully.

5.2 Node Removal Algorithm

When a node wants to remove a connection it selects the worst neighbor among the neighbors it currently has. The worst neighbor has the smallest goodness value. The goodness is the sum of the neighbor connection's hit values and relayed hits.

$$Goodness = hits + relayedhits \tag{1}$$

The Node Removal Algorithm (Algorithm 5.2) is described as follows.

Algorithm 5.2 (NodeRemovalAlgorithm)
Input: Connections $C = \{c_1, ..., c_n\}$, where c_is are node's neighbor connections.
Output: Removes the worst connection.
c = null
$lowestGoodnessValue = \infty$
for i=1 **to** $\mid C \mid$ **do**
  g = Hits(c_i)+ RelayedHitsSum(c_i)
  if $g < lowestGoodnessValue$ **then do**
    $c = c_i$
    $lowestGoodnessValue = g$
  end if
end for
if $c \neq$ null **then do**
  DisconnectConnection(c)
end if

The function Hits(*connection*) returns the connection's hit values and the function RelayedHitsSum(*connection*) returns the sum of relayed hits

of the connection's neighbors. The method DisconnectConnection(*connection*) removes the connection to the neighbor.

5.3 Overload Estimation Algorithm

There is no predefined number for the connections the node should maintain. Thus the connections are added and dropped based on the amount of traffic going through the node. The Overload Estimation Algorithm compares the traffic meter value calculated in the ConnectionManager to the predefined traffic limit values. There are upper and lower traffic limits which set the range where the traffic amount should be. If the traffic is more than the predefined upper traffic limit, one connection is dropped by using Algorithm 5.2. If the traffic is less than the lower traffic limit it tries to add a new connection using the Algorithm 5.1. At the end, the algorithm resets the traffic meter by setting its value to zero.

Algorithm 5.3 (OverloadEstimationAlgorithm)
Input: Connections $C = \{c_1, ..., c_n\}$, where c_is are node's neighbor connections, value of the traffic meter *meter* in kilobytes, value of the upper traffic limit *upperLimit* in kilobytes and value of the lower traffic limit *lowerLimit* in kilobytes.
Output: Establishes a new connection or removes one connection.

```
overloadFlag = false
if meter > upperLimit ∧ | C |> 1 then do
    overloadFlag = true
    NodeRemovalAlgorithm(C)
end if
if meter < lowerLimit then do
    NodeSelectionAlgorithm()
end if
meter = 0
```

The variable *overloadFlag* is true if the traffic amount is greater than the traffic limit. In that situation the node does not accept any new connections.

5.4 Overtaking Algorithm

The Overtaking Algorithm is used to optimize the topology. The purpose of the algorithm is that the node moves in the network closer to the nodes which provide it a lot of replies by overtaking the current connection. The node does not directly connect to a neighbor of the resource providing node but only closer step by step and that way makes sure that it does not lose good nodes on the path.

The idea is that when a reply message arrives to the querier it updates the hit value of the replier node and updates the local information concerning the relayed hits of the neighbor of the connection from which the node got the reply message. Then if the connection's hit value is bigger than 1, i.e. the node has got more than one message from the neighbor, the node checks whether the connection has a neighbor node whose proportion of the sum of all neighbors' relayed hits and connection's hits is more than the defined overtaking percent. For example if the overtaking percent is 60% it means that if there is the neighbor of the connection which has forwarded over 60% of all reply messages the node has received from the connection then the node establishes a new connection to that node and drops the current connection.

The advantages of the algorithm are that the distances of the nodes which use others' resources are shorter than in randomly connected networks. The algorithm creates clusters gathering close to its center the nodes which provide a lot of resources used by other nodes.[2, 3]

Algorithm 5.4 (OvertakingAlgorithm)
Input: Overtaking percent *overtakingPercent*, node's neighbor connection c, c's neighbors $N = \{n_1, ..., n_n\}$ and c's hit value $hitValue$.
Output: Node has overtaken a neighbor if some neighbor's neighbor is better for the node.

```
if hitValue > 1 then do
    sum = 0.0
    biggest = overtakingPercent/100.0
    bestNeighbor = null
    sum += Hits(c) + RelayedHitsSum(c)
    for i=1 to | N | do
        hitValue = RelayedHits(n_i)
        proportion = hitValue/sum
        if proportion ≥ biggest then do
            biggest = proportion
            bestNeighbor = n_i
        end if
    end for
    if bestNeighbor ≠ null then do
        if EstablishConnection(bestNeighbor) then do
            DisconnectConnection(c)
        end if
    end if
end if
```

The function Hits(*connection*) returns the connection's hit values, the function RelayedHitsSum(*connection*) returns the sum of the relayed hits of the connection's neighbors and the function RelayedHits(*neighbor*) returns the relayed hits of the neighbor. The function EstablishConnection returns

true, if establishing a connection succeeded. The method DisconnectConnection(*connection*) removes the connection to the neighbor.

6 Conclusion

The Chedar peer-to-peer middleware provides a decentralized architecture for P2P applications. The topology of the Chedar network is self-organized by the topology management algorithms and different search algorithms can be used for discovering the resources. Future work of Chedar includes further development of the topology management algorithms and NeuroSearch resource discovery algorithm to optimize the search process as well as the mobile peer-to-peer application development on top of Mobile Chedar.

References

[1] L. A. Adamic, R. M. Lukose, and B. A. Huberman. Local search in unstructured networks. In *Handbook of Graphs and Networks: From the Genome to the Internet*, pages 295–317. Wiley-VCH, 2003.

[2] A. Auvinen. Topology management algorithms in chedar peer-to-peer platform. Master's thesis, University of Jyväskylä, February 2004.

[3] A. Auvinen, M. Vapa, M. Weber, N. Kotilainen, and J. Vuori. New topology management algorithms for P2P networks. Unpublished.

[4] Cheese factory. http://tisu.it.jyu.fi/cheesefactory.

[5] Gnutellahosts. http://www.gnutellahosts.com/.

[6] B. J. Kim, C. N. Yoon, S. K. Han, and H. Jeong. Path finding strategies in scale-free networks. *Physical Review E*, 65, 2002.

[7] N. Kotilainen, M. Vapa, A. Auvinen, T. Keltanen, and J. Vuori. P2PRealm - peer-to-peer network simulator. Unpublished.

[8] N. Kotilainen, M. Vapa, A. Auvinen, M. Weber, and J. Vuori. Peer-to-peer studio - monitoring, controlling and visualisation tool for peer-to-peer networks research. Unpublished.

[9] N. Kotilainen, M. Vapa, M. Weber, J. Töyrylä, and J. Vuori. P2PDisCo - java distributed computing for workstations using chedar peer-to-peer middleware. In *Proceedings of the 19th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2005)*, 2005.

[10] N. Kotilainen, M. Weber, M. Vapa, and J. Vuori. Mobile chedar - a peer-to-peer middleware for mobile devices. In *Proceedings of the Second International Workshop on Mobile Peer-to-Peer Computing (MP2P05)*, pages 86–90. IEEE Press, 2005.

[11] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[12] N. A. Lynch. *Distributed Algorithms*. Morgan Kauffmann Publishers, 1996.

[13] S. Nazarko. Evaluation of the data fusion methods using kalman filtering and transferable belief model. Master's thesis, University of Jyväskylä, November 2002.

[14] A. Oram, editor. *Harnessing the Power of Disruptive Technologies*. O'Reilly, Sebastopol, CA, 2001.

[15] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of First International Conference on Peer-to-Peer Computing (P2P'01)*, pages 101–102, 2001.

[16] M. Vapa, N. Kotilainen, A. Auvinen, H. Kainulainen, and J. Vuori. Resource discovery in P2P networks using evolutionary neural networks. In *International Conference on Advances in Intelligent Systems Theory and Applications (AISTA 2004)*, November 2004.

[17] M. Weber, J. Vuori, and M. Vapa. Advertising peer-to-peer networks over the internet. *Radiotekhnika*, 133:162–170, 2003.