Institute of Parallel and Distributed Systems
Department Distributed Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Studienarbeit Nr. 2408

Concepts and Mechanisms for Consistent Route Transitions in Software-defined Networks

Benjamin Gayer

Course of Study:	Computer Science
Examiner:	Prof. Dr. rer. Nat. Dr. h. c. Kurt Rotherml
Supervisor:	Dr. rer. nat. Frank Dürr

Commenced:	12.11.2012
Completed:	14.05.2013
CR-Classification:	C.2.0, C.2.2, C.2.4, C.2.6

Index

List of figures

Abstract

Software-defined Networking (SDN) [1] is a big trend in network research and industry. The key idea of SDN is to separate the control and the forwarding functionality. In conventional networks the firmware on the switches determines how the switches handle packets, so that they treat all packets in exactly the same way. This leads to static networks, that can not adapt to changing requirements. In Software-defined Networks a (logically) centralized controller enables the network administration to change the routing simply by updating the controller. The controller then can change the flow table entries of a subset or even all the switches in the network. There is no longer the need to update every switch separately. SDN is in general used for highly adaptive routing to fit the requirements of dynamic load, frequent topology changes, migration of virtual machines and hosts.

This work is about consistent route updates in Software-defined Networks. Two classes of consistency have to be distinguished. The first one is eventual consistency, that means during the update inconsistency's can occur, but the final state will be consistent. The second one is strict consistency, here the routes are always consistent, even during the update process. Inconsistent updates can lead to security issues, loss of connection, inaccessibility and many other problems. In current networks updates are necessary to fit the frequently changing requirements.

The problem with (strict) consistent updates in SDN is that there are no atomic updates because the switches are inherently distributed. And even if there would be such an update, it would affect packets in transit. Therefore the goal is to avoid transient route inconsistencies like “black holes” and loops.

There are already a couple of update strategies for SDN which result in consistent updates, but all of them are limited in some way, for example some can just be used for OSPF or BGP. There is also one approach by Reitblatt et al. [3], that is not limited. This strategy is a two phase update which leads to a “per packet consistency”. The old route and the new route are installed at the same time, so that every packet is on a consistent route (the old route, before the update, or the new one, after the update). This approach has an overhead in terms of storage-use, because the new route exists at the same time as the old one and also needs rewriting of the packet headers to signal the phase. But the storage capacity of switches is limited and so a doubling of forwarding table space is a high burden. The approach shown in this work is more light-weight and requires no change of header fields and no additional forwarding table space or any other modifications of the switches.

The key idea is to update the switches backwards according to the new route. That means the first switch that is updated is the predecessor of the destination and the last one is the source. So packets will follow the old route until they reach an already updated switch which will forward them along the new route. This is also true for packets in transit. On the downside this approach can in some situations “only” lead to the eventual consistency, which is a result of the underlying network model (asynchronous communication). To achieve strict consistency it would be necessary to avoid certain updates or to change the underlying network model.

1. Introduction

In the last years the requirements for networks changed, such that the traditional network architecture no longer can satisfy the needs of the networks. In the past, networks were designed to fit client-server communication. In the traditional network architecture switches are forwarding packets according to there pre-installed firmware. All packets are handled in the same way by the switch. That is the reason why it is a problem for common networks, which are static, to adapt to dynamic changes.

Today we are dealing with different requirements. Cloud-computing for example needs dynamic distribution of storage and processing power. Carriers today are facing a fast growing need for more bandwidth and mobility.

It is a very complex and challenging task to adapt the traditional network architecture to todays tasks. Complex protocols are developed to achieve this. To implement a new policy it can be necessary to update hundreds or thousands of devices manually. This is of course very time-consuming, so it can for example take a few hours to migrate a single virtual machine in a data center.

To face the new requirements, Software-Defined Networking (SDN) [1] was developed by the Open Networking Foundation (ONF). SDN is a network architecture, where the control functionality (done by the controller in software) is decoupled from the forwarding functionality (performed by the switches in hardware). In this way it is possible to abstract the underlying infrastructure, such that applications can treat the network as a logical unit. The controller is (logically) centralized and has a global view of the entire network, in contrast to traditional networks, where on each switch a distributed algorithm is running independently. Network administrators now can configure the network just by changing the software running on the controller and do not need to update every single device in the entire network separately, but only the controller. This makes SDN highly flexible so that it can face todays requirements.

The OpenFlow protocol [2, 5] offers a standard for the communication between the controller and the switches in SDN. OpenFlow was the first standard and is already accepted and supported by academia and industry.

As described above, SDN provides a powerful tool for adapting networks to changing requirements. Though SDN assumes a logically centralized controller the whole network is in fact still a distributed system. Therefore updates can not occur at precisely the same time on every switch. Even if the initial and the final state are consistent, this may lead to inconsistent updates, which result in transient “black holes” or transient loops. If an already updated switch forwards a packet to a switch that was not part of the old route, then this new switch has no entry in its forwarding table and therefore does not know what to do witch this packet – the packet is in a black hole – which means the packet is lost. Loops cause packets to circle in the network, so that they may not reach their destination before they are dropped because their time to life equals zero.

But even if there would be an atomic update such that the forwarding tables of all switches become updated at the same time, such an update would still hit packets that are in transit.
There are several approaches that address this problem, for example one by Reitblatt et al. [3] and another one by MCGeer [4]. The approach by Reitblatt et al. is a two phase update, that means during the update the old route persists and a second route is introduced, this approach modifies the packet header fields to signal to which phase (route) a packet belongs. This approach needs also a lot of forwarding table space on the switches, which is limited. The approach shown here does not need a modification of header fields and needs less forwarding table space.
The second approach by MCGeer needs much less forwarding table space than the one by Reitblatt et al. and also no modification of header fields, but it creates a lot of additional traffic around the controller, because the control plane is exploited to forward data during the update. The new approach described here does not use the control plane to forward packets, therefore it creates no additional traffic around the controller.
In this work a new approach is shown that avoids black holes and leads to eventual consistency. In some cases the approach can lead to strict consistency as well, but without modification of the basic approach this is not true in general, because it is possible that there occur transient loops during the update process. The idea is to update the route beginning with the destination, backwards to the source. In contrast to previous approaches this one will not need changes of header fields. For each packet a guarantee can be given, that the packet will follow entirely the old path (before the update), entirely the new path (after the update), or it will start on the old path and then, after is has reached the first updated switch, continue its way through the network on the new path (during the update). Furthermore it is guarantied that after a packet has reached an updated switch it will not reach a switch that is not updated.
In the following “OpenFlow” and SDN are described. Then there is an introduction to unicast-routing in general, followed by an overview of related work. In the third chapter the system model, this work is based on, is introduced. Furthermore it contains a formal description of the problem. The forth chapter contains the solution of the described problem and a formal proof of the correctness of this solution. Chapter five describes the implementation, which solves the update-problem, in the way described in chapter four. The implementation was tested and the results, as well as informations about the test-setup and the used metrics, can be found in chapter six. All this is finally summarized in chapter seven.

2. Background and related work

2.1 Unicast-Routing

In this chapter an overview about unicast routing strategies is provided, note that just routing in electronic data networks using packet switching technology is concerned. In packet switching networks packets are forwarded from the source to the destination via intermediate nodes which can be for example routers or switches. The routers and switches (in the following just the term switch will be used) maintain forwarding tables that contain the information on what outgoing line a packet has to be forwarded to reach its destination. Routing is the task to find a path from the source to the destination. A routing decision can be made separately for each packet or can be done once for a whole stream (often called "flow") of packets.
In a unicast network a packet is forwarded from one source to exactly one destination. This is in contrast to multicast networks where one packet can be forwarded to multiple destinations. Broadcast is a special case of multicast where one packet is forwarded to all nodes within the network.

Two kinds of routing strategies can be distinguished in general:

1. Static routing (also called non-adaptive routing) is using pre-computed routing tables and can therefore only be used in static networks or in networks, where changes are occurring very rarely.

2. Dynamic routing (also called adaptive routing) uses routing tables that become generated automatically by routing protocols, this makes it possible to route packets in dynamic networks, where topology changes are occurring frequently.

In the following the focus is on dynamic routing, because SDN, which is the topic of this work, is a design that was invented to face the flexibility problems in dynamic networks.
There are three different strategies for dynamic routing:

1. **Centralized routing** uses a (centralized) Routing Control Center (RCC). In general centralized routing is not suitable for distributed systems because it provides a single point of failure, but because SDN assumes a logically centralized controller it is worth a closer look. The main advantage of centralized routing is that, due to the global view of the RCC, it is possible to compute optimal routes. Routes can be computed for different metrics, for example minimum delay by using Dijkstra's algorithm. Another advantage is that not every node has to use resources to compute its routing table.

Depending on the algorithm (and the networkt topology) it is also possible that a centralized algorithm reduces the comunication costs (it is the number of messages send) for creating the routing table.
The main disadvantage is, that the RCC is of course a single point of failure and if it fails no routing updates can be done, but messages are still forwarded based on the (outaged) routing tables of the switches. There is also a high communication effort for the RCC which has to send routing table updates to every switch and has to receive messages containing information about topology changes. Another problem closely related to the topic of this work is inconsistent updates. Inconsistent updates can occure in centralised routing, because the central computed routig tables have to be send to the switches. It is not possible to ensure that every switch will receive or install the update at the same time.

2. **Isolated routing** is a strategy, where every switch does its own routing. There is no exchange of routing information. A popular representative is the “Backward-Learning-Algorithm” which is introduced now.
   If a switch has no entry in its routing table for a packet it will forward the packet on all outgoing lines, this is called flooding. Flooding creates a huge message overhead, but ensures that the packet will reach its destination (If it is reachable) and it also finds the shortest path(s). In addition to this first rule, there is another rule: If a switch receives a packet it also learns a path to the source with the length coded in the header of the packet. If the learned route is better (the length of the path is smaller) then a new entry is added to the routing table. In this way not every packet has to be flooded because after every flooding all switches will know a route to the source. That means the switches are learning routs over time.
   The problems with this approach are, at first, the overhead created by flooding and that the network is not learning about changes in the network, if they decrease the performance. The last problem is solved by introducing a time to live for the routing table entries, so a switch will forget about routes. This introduces, of course, a parameter which is critical for the performance of the algorithm. The first problem can not be solved, but its effect can be reduced by limiting the flooding and doing a more clever flooding (for example do not send messages to switches from which you already received the message or only flood a message once, even if you receive it multiple times).

3. **Distributed routing** is an approach, where every node periodically exchanges routing information with its neighbors. This information is then used by each node to set up its own routing table. There are two different types of algorithms for this approach.

The first one is the “distance-vector routing”. To every connection between two nodes a number representing the costs is assigned. Each node knows the costs to its (direct) neighbors and periodically sends this routing information (distance-vectors) to its neighbors. If a node receives a distance-vector from its neighbor, it can compute the communication costs to all neighbors of the sender via the sender. In this way the locally measured costs are propagated through the network. Improvements in the communication costs are propagated fast, but if the communication costs become worse, then the algorithm suffers a problem called “count to infinity”. If a node A fails then its neighbors will notice that and set the communication costs to this neighbor to infinity. This information is propagated to their neighbors, but ,unfortunately, the neighbors know a better path to A via another node and the neighbor of A. This is in fact a cycle! That means that the direct neighbors of A will learn a new path to A which is cheaper than infinity but does not exist in reality. Of course, finally, the communication costs within the cycle will increase to a value that is representing infinity, but this takes some time.
The second algorithm is the link-state routing. Each node measures the communication costs to its direct neighbors and then sends this information as a link-state-packet to each node in the network, this creates a global view of the network. To send the link-state-packets flooding is used. If a node has received a link-state-packet from every other node within the network it can compute its routing table based on this global view. The downside is that every node computes exactly the same routing table, which is of course a waste of processing power. Furthermore this approach does not scale for large networks.

2.2 SDN/OpenFlow

Software-defined Networking (SDN) [1] is predicted to be one of the biggest network-trends in 2013, many big companies have announced SDN enabled hardware products for this year. In the SDN network architecture the control plane is decoupled from the forwarding functionality and is directly programmable. This is achieved because the network-control is done by a logically centralized controller, which is connected to all switches within the network and can program the forwarding tables of this switches. Forwarding is then performed by the switches, according to the forwarding table entries, at line rate. This makes it possible for applications to abstract the underlying infrastructure and to treat the network as a logical unit. The network architecture is shown in figure [1]. The Application can access the SDN controller via an API, the control is done by this software-based controller, which has a global view and can therefore be used to compute optimal routes for a chosen metric. The centralized controller enables IT to change the networks behavior in real time and provides the full control over the network to the user. The fact that the controller is now programmable by the user, and not only by software updates provided by the hardware vendors, leads to an unprecedented flexibility of the networks.
Because SDN assumes a centralized controller which sends the routing tables to the switches, special switches are needed, that are able to communicate with the controller.
The OpenFlow protocol is the first standard communication interface between the control and the forwarding plane, that means it describes the communication between the controller and a switch. OpenFlow is based on Ethernet switches and allows it to add or remove entries in the forwarding tables of switches. As the name indicates OpenFlow uses the concept of flows, that means each connection between two endpoints is described by a unique set of parameters like, MAC address, IP address or port number of source and destination. The usage of flows allows it to route the communication between two endpoints on different paths, for example based on QoS (Quality of Service) it would be possible that the data of a delay sensitive application is routed on a path with a low end-to-end delay, while a data transfer, which needs a high bandwidth, is routed on a path that provides a large bandwidth. Current IP based routing does not provide this flexibility, because it routes all the communication between two end points along the same path. In the meantime OpenFlow is well accepted and supported by academia and industry. Note that OpenFlow provides an interface which can be used for the communication between controller and switches, but it provides no functionality like to ensure the order of the execution of an update. Therefore OpenFlow is no solution to the basic problem of updates in distributed systems, which is the fact that it is impossible to ensure that all switches receive the update at the same time.

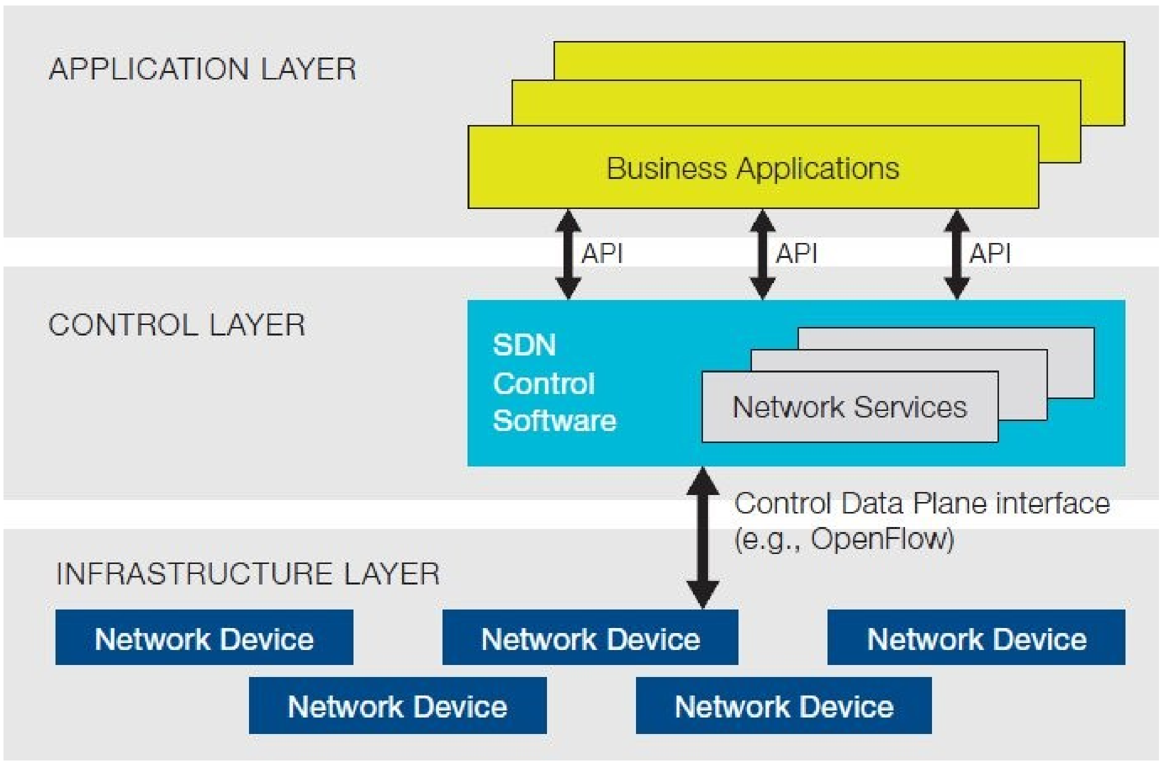

Figure [1]: The SDN Architecture. Source: [1] on page 7

2.3 Related work

There has been a lot of research around many aspects of SND and OpenFlow, there are also some papers that are closely related to the topic of this work. Two of these papers are introduced in the following section, because of their importance for this work. The first one is interesting because the approach described within it gives almost the same guarantees as the approach described in this work. The second one can be seen as “state of the art” and is therefore the approach this work will be compared with.

The first paper that is discussed here is “A Safe, Efficient Update Protocol for OpenFlow Networks” by Rick MCGeer [4]. The approach described by MCGeer exploits the SDN architecture in which a centralized controller is connected to each node via a control link, that is originally only used for transmitting control messages.
The update protocol consists of four epochs. In the first epoch the old ruleset is installed on the switches and all packages are following the old path. In the second epoch an intermediate transfer function is send to the switches. According to this intermediate transfer function a switch is forwarding packets in the same way as in the old ruleset, if the new ruleset is equal to the old one. If the new ruleset differs from the old one, the switch forwards all packets to the controller. Every switch sends a completion signal to the controller if it has installed the new ruleset. If the controller has received a completion signal form each switch it waits for the maximum network delay and then epoch three begins. In epoch three the controller sends the new ruleset to all switches, which send a message back that is indicating, that they are now forwarding packets according to the new ruleset. If the controller has received such a message from every switch, it will again wait for the maximum network delay. After that the update is completed.
The maximum space that is required on the switches is the maximum of the three rulesets (the old, the intermediate and the new ruleset). The intermediate step is necessary because for any switch it is unclear if it is forwarding packets according to the new or the old ruleset, during the update process.
This approach gives the guaranty that a packet is handled by a consistent set of rules, either completely by the old ruleset, or completely by the new ruleset. If a packet is once forwarded under the new ruleset, all following packets are forwarded according to the new ruleset. This leads to a “per flow consistency”, a flow is either entirely forwarded under the old ruleset, or entirely under the new ruleset, or the prefix of the flow is forwarded according to the old ruleset and the suffix is forwarded according to the new ruleset.

The approach shown in this work has similar guarantees. A packet is either following entirely the old route, entirely the new route, or it starts on the old route and after it reaches the first updated switch, it is forwarded from there on the remaining part of the new path. Furthermore the approach described in this work introduces the missing knowledge, that makes the intermediate transfer function necessary in MCGeer's approach.
The downside of MCGeer's approach is obviously the high traffic that is generated around the controller, which is not only sending and receiving control messages, but also actively forwarding data packets (the ones that are forwarded to the controller during epoch two). In contrast the approach shown in this work is not using the control layer to forward packets and therefore the controller is not involved in the forwarding.

The second paper that is introduced now is “Abstractions for Network Update” by Reitblatt et al. and was published on the SIGCOMM'12. The approach avoids the inconsistencies that can occur during the update by using a two phase update strategy. The old route persists during the update and an additional flow, along the new route, is created. Therefore it is necessary to modify the header fields to indicate to which phase a packet belongs. At first packets follow the old route, then the controller creates the new flow along the new route. After this the ingress switch is updated to change the header field (typically the VLAN tag is used) to indicate that all following packets belong to the second phase. All packets that are tagged prior to the ingress switch update follow the old route to their destination, all packets that are tagged after the update of the ingress switch follow the new route. This approach provides per packet consistency, which means that each packet is handled by exactly one globally consistent state. This means each packet is processed either on the old path before the update, or on the new path after the update, but never on a mixture of them.
A strength of this protocol is that the properties of a path are preserved during the update, for example if the path is free of forwarding loops before and after the update, it is guaranteed, that there will be no loops during the update. On the downside this approach needs forwarding table space for both routes, the new and the old one, at the same time and it is necessary to modify the header field. The approach described in these works requires no modification of the header fields and needs less forwarding table space.

3. System-Model and Problem Statement

In the following a description of the System-Model that is required by the algorithm, described in this work, is given. The algorithm assumes an asynchronous system, that means there is no maximum delay for messages that are sent. The only guaranty that is given an asynchronous system is that the messages arrive eventually – that means also that no messages are lost, which is important for an update-protocol. Furthermore the only other assumption is that we assume FIFO (First In First Out) channels, this is required for the use of TCP in practice. Note, that even if FIFO channels are used, that there is no guaranty that the FIFO order consists during an update. For example the old route could be one with a high delay, while the new route has a low end to end delay. In this case it would be possible, that the first packet send via the new route arrives before its predecessor, which is the last packet send via the old route. The algorithm assumes an SDN, the components used within such an SDN are introduced in the following.

3.1 System-Model

This section is about the network, more precisely the hardware components used within a SDN. There are three components: Hosts, multilayer switches and the controller. The hosts are connected (they can exchange information/data) via the multilayer switches with other hosts. The multilayer switches are forwarding the packets, send by the hosts, to their destinations. The routing is done by the controller which calculates the routes through the network and sends the resulting forwarding tables to the switches. In the following section the single components are considered in detail.

Hosts are participants in a network, (Assuming the client-server model) they can be clients or servers . A client is a hardware component that is using services that are provided by another hardware component, which is called server. These two separate hardware components (Client and Server) are communicating with each other via the network. In addition to the client-server role a host can also be a node in a peer-to-peer system. A peer is a network participant that is a server as well as a client. In general a host is a communication endpoint in a network.

Multilayer switches are a group of switches, that are using more packet information for the switching than ordinary switches, which only use the information provided on layer two (data link layer) of the OSI model. A Switch is a network device that forwards incoming packets to their destination based on the information provided in the header of the packets. Every switch has its own forwarding table, which is of course limited in terms of storage capacity.

The **Controller** is something like the brain of a SDN, it is calculating the forwarding table for each switch and sends these tables to the switches. To fulfill this task the controller has a direct link to each switch, these links are called control links and are used to exchange control messages. There are approaches that use the control links also for forwarding tasks, but the approach described here is only using them for control messages. The controller provides two interfaces, the first one is the so called southbound interface which is responsible for the communication between the controller and the switches. As described above there is already a standard for the communication between controller and switch, which is the OpenFlow protocol. The second interface is the northbound interface, which defines the communication between controller and applications. For example the controller exposes information about network topology or traffic to the application. Until now there is no standard for this interface defined, for this work floodlight was used.

Figure [2] shows an example network with 5 hosts, 3 switches and one controller.

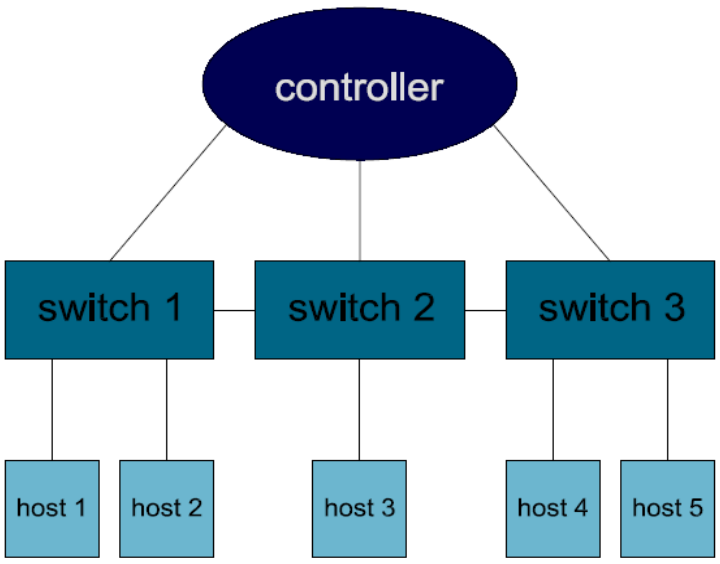

Figure [2]: An example topology showing a SDN with 5 hosts, 3 switches and one controller

3.2 Problem Statement

As described above the problem with updating routes in SDN is, that it is hard to guarantee consistency during the update, because each update hits packets that are already in transit. In order to perform an update routing tables on distributed switches have to be updated by the (centralized) controller. Without a maximum message delay the only possibility to ensure the order in which the switches are updated is to update one single switch and to wait for its confirmation. But this strategy is no solution to the basic problem, which is how to determine the order for a consistent update. The next chapter shows an algorithm that solves this problem. In the following the problem is formalized.
A route is defined by a sequence of switches s, these switches can be numbered, starting with the first switch $(switch_1)$ that receives a packet. To the last switch $(switch_n)$ the number n is assigned, this switch is the one directly before the receiver or it is an egress switch (switch that forwards packets out of the network). A route update u is a sequence of k instructions m_j (j ϵ [1..k]) that converts one route r_{old} into another route r_{new} . That means one sequence of switches is converted into another sequence of switches. The problem of a consistent update is to order the instructions m_j in a way such that no black holes and no loops occur. A black hole occurs if a packet is forwarded to a switch that has no flow-table-entry for the flow the packet belongs to. This results in the drop of the packet. A loop is a sequence of switches, in which at least one switch occurs more than once. That means a packet that is forwarded according to such a sequence will reach at least one switch more than once. The packet is in a circle. Such a circle can be a transient loop, that occurs during the update but is not present in the final route after the update, but it can also be a permanent loop, that is also present in the final route after the update is finished.

4. Conceptual Design

This chapter provides detailed information about the new algorithm that is the topic of this work. At first the algorithm is introduced in an informal way, than a formal description follows. After the formal descriptions it is shown that the update can cause transient loops and it is proven, that the update protocol avoids black holes.

4.1 The Algorithm

The key idea of the algorithm is to update the new route backwards. That means if there is an old route that has to updated to a new route, then the algorithm will at first update the last switch on the new route. After the last switch is updated the one before the last one is updated and so on until the update reaches the first switch. If a switch is part of the new route as well as of the old route, then its old routing table is replaced by the new one (this step helps to save routing table space and avoids the modification of header fields that is necessary in a two phase update, as the one by Reitblatt et al.). If a switch is part of the new route, but not part of the old route, than a new entry is added to its routing table.
More formal:
Given two sequences of switches s, that belong to two different routes r_{old} and r_{new} , the one sequence can be converted into the other one by executing the following algorithm:
For each switch s on the new route there is one instruction m. There are two types of instructions, the first one is updating an existing flow-table-entry. The second one is adding a new flow table entry to the forwarding table of a switch. The first kind of instruction is used for switches that belong to both sequences. The second one is used for switches that belong only to r_{new}. The instructions are changing the forwarding-table-entries of the switches according to the sequence of r_{new}. The instructions m_j are ordered in a way that their order matches the reverse of the sequence of switches s of the new route r_{new}. If the sequence of switches is numbered consecutively by 1 to n, then the instructions are ordered starting with m_n downwards up to m_1.
If the instructions are executed in this order, then the resulting sequence of switches s is free of loops. Furthermore during and after the update process there will be no black holes. But it is possible that transient loops occur during the update process. Transient loops are discussed in detail in the following section. Note that during and after the update in each flow-table there is at most one entry per flow.

4.2 Transient Loops

During the update the old route will not persist. The old route is much more something like a bypass (or a sequence of bypasses) to the new route, that is used by packets that are in transit on the old route during the update. This also means, that during the update it is hard to tell what way each packet takes. A packet may start on the old route and continue its way completely on the old route, or it starts on the old route and continues on the new route. In general the following holds: After a packet has once been forwarded on the new route it will no longer be forwarded along the old route, because all switches below the path are already updated. However there is an exception to this general behavior. The update strategy described here prevents no transient loops. Transient loops are loops that occur during the update process but do not persist and are therefore not present after the update. Transient loops occur if the new route connects (direct or indirect!) two switches that are connected (direct or indirect) on the old route in reverse order. If $switch_a$ forwards packets to $switch_b$ on the old route and $switch_b$ forwards packets to $switch_a$ according to the new route, then packets that are in transit between this two switches during the update of $switch_a$ will be forwarded back to $switch_a$ and therefore circle exactly one time. Figure 2 shows an example for a topology that leads to a transient loop during the update. Figure 2 a) shows the topology before the update, figure 2 b) shows the loop during the update and figure 2 c) shows the resulting topology after the update.
According to the update protocol at first $switch_d$ (in the following per switch just one capital letter is used to address it.) is updated, in this case no changes occur. In the next step B is updated to forward packets to D instead of forwarding them to C. In this way an intermediate route is created that forwards Packets from A to B and from B to D. After that, C's routing table is updated to forward packets to B. This step creates the transient loop! Packets that are in transit between B and C during this update step will be forwarded from C back to B. That means this packets circle exactly one time. In the next step A is updated and the update is finished.
Always, if a transient loop is generated during the update, packets can circle in this transient loop at most once and all packets that enter such a transient loop will circle there exactly one time.
The Algorithm described in this work provides no solution to this problem.
However it is important, that even if the algorithm produces transient loops the final route will always be free of loops, because there is always at most one flow-table-entry per flow at each switch at each time. With one flow-table-entry it is possible to create a loop, but in this case the destination is not part of the route with the loop. This case can not occur, because we start with the destination and update from there.

4.3 Black holes

The update protocol described above will always lead to black hole free routes, that means every switch that receives a packet always knows to which switch it must forward the packet. This is not only valid for the start and the final state, but also for all intermediate occurring states. The proof of this is shown below.

Let the switches on the new route be consecutively numbered, where the start has the number 1 and the destination is labeled by n.

At the beginning there is just the old route, which is by definition black hole free. (Note: Even if there is no route at the beginning, the route created is black hole free)

According to the update protocol, an entry to the flow table of $switch_{n-1}$ is added, so that arriving packets, which belong to the updated flow, will be forwarded to $switch_n$. If there already is an entry in the flow table, this entry is overwritten (changed).
There are 3 cases that have to be distinguished.

1.) $Switch_{n-1}$ is the start:
That means, there is now a new route from the start (n-1) to the destination (n) and every switch on this route has a flow table entry for the flow. In addition to that an modified version of the old route exists. It is not exactly the old route, because $switch_{n-1}$ is of course also the start of the old route and the entry in its flow table has been changed. That means new packets will follow the new route, which also means that there are no new packets entering the old route. Packets that have entered the old route before the update will reach the destination on the old route. So there are neither black holes on the new route, nor on the old route.

2.) $Switch_{n-1}$ is a switch on the old route, but not the start.
Like in the first case, packets arriving at $switch_{n-1}$ are forwarded to $switch_n$ and packets that already have passed $switch_{n-1}$ will continue their way on the old route to $switch_n$. New packets will be forwarded on the old route until they reach $switch_{n-1}$ then they will follow the new route to $switch_n$. As described above the old route is considered to be black hole free, furthermore the new route is black hole free.

3.) $Switch_{n-1}$ is not on the old route.
That means no other switch will forward packets to $switch_{n-1}$ and all packets will follow the old route. The change made has no effect to the network so far. The old route is completely unaltered and therefore black hole free.

The next step is to update all the other switches along the new route in reverse order. That means the next switch updated is $switch_{n-2}$. In general the flow table of $switch_{i-1}$ (1 < i < n) is updated, so that there is a connection from $switch_{i-1}$ to switch $switch_i$. Note there is also a connection from $switch_i$ to $switch_n$. Again 3 different cases must be distinguished.

1.) i=2: In this case $switch_{i-1}$ is the start.
As described above $switch_{i-1}$ is connected to $switch_i$, which is connected to the destination. Therefore the new route is completely established. All new packets will follow the new route. Packets on the fly will follow the old route until they reach a switch that belongs to the new route, after that they will follow the new route. If there is no switch, that belongs to the old route as well as to the new route, then packets on the fly will follow the old route to the destination.

2.) $Switch_{i-1}$ is a switch on the old route and not the start (i > 2).
In this case all packets arriving on $switch_{i-1}$ are forwarded to the destination completely on the new route. Packets that already have passed $switch_{i-1}$ are following the old route, until they reach another switch, that is part of the new route and already updated. then they will follow the new route to the destination.

3.) $Switch_{i-1}$ is a switch, that is not on the old route.
As seen before this update is not affecting the old route. The new route stays also black hole free.

The last step of the update process is to delete all flow table entries, for the updated flow, on all switches that belong to the old route, but not the new route. This could cause black holes, if these entries are deleted before all packets on the fly are forwarded to the destination (that would be the maximum propagation time of the old route). If we avoid this, then there are no black holes at any time during the update process and of course the final state is black hole free.

5. Implementation

The algorithm introduced above was implemented and tested on a testbed. The topology of this testbed is shown in figure [3]. This chapter contains a description of this testbed and explains how the algorithm was implemented. The testbed consists of 10 Open vSwitches and 2 virtual machines, each running 4 hosts (8 hosts in total), as controller Floodlight was used. Because OpenFlow is the standard for the communication between Switches and Controller (Southbound Interface), it was used in the testbed. In the following a short description of the floodlight controller is given, followed by an explanation, how the algorithm was implemented.

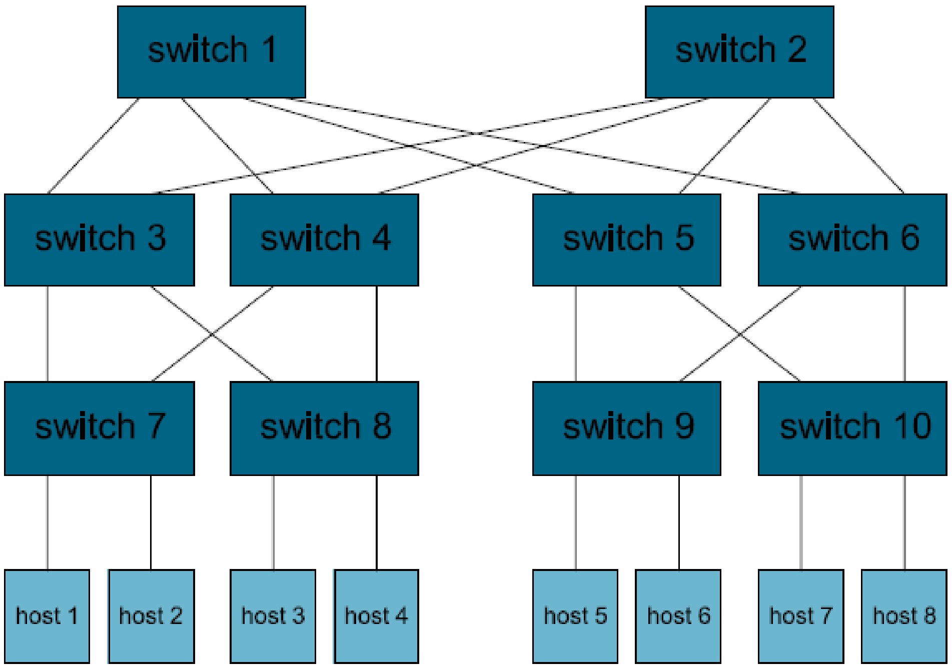

Figure [3]: The topology of the Testbed that was used to test the algorithm.

5.1 The Floodlight Controller

Floodlight is a controller, but it also provides a collection of applications build on top of the Floodlight Controller. While the Floodlight Controller, which is written in JAVA, provides functionality to control an OpenFlow network, the applications are the so called “Northbound Interface”, which are used for the communication between controller and other applications that are using the controller in order to do their tasks. In fact Floodlight provides two different interfaces. The first one is the Module Interface the second one is the REST Interface. This architecture is shown in figure [4].
The Module Interface supports proactive as well as reactive routing. Proactive routing means that the entries into the flow tables of the switches, that are needed to forward the packages correctly, are added to the flow tables before the first packet is sent. Reactive routing means that the first switch that receives a packet that belongs to a flow, it has no entry for in its flow table, forwards this packet to the controller, which then sets up the forwarding tables of all switches. In contrast to this the REST Interface is less powerful and only supports proactive flow programming. For testing the algorithm it is sufficient to use proactive routing, therefore only the REST Interface is introduced.
Representational State Transfer (REST) is a software architecture that is used in distributed systems. The REST architecture assumes a client-server-model in which the client sends a request to a server that processes this request and returns a reply. REST is resource-oriented, which suits it well for the task, because the single switches, that the user or an application (the client) wants to access, match the resources and the controller is the equivalent to a server. To manipulate resources (to modify flow tables) HTTP methods like GET, POST, PUT, etc. are used. Because of this it is possible to modify the flow tables with the following command line code:

```
curl -d '{"switch": "00:00:00:00:00:00:00:05",
 "name":"flow-mod-11",
 "priority":"32768",
 "ingress-port":"2",
 "active":"true",
 "actions":"output=3"
 }' http://localhost:8080/wm/staticflowentrypusher/json
```

The command line tool “cURL” posts HTTP requests, which be used to access the REST API. The json payload tells the Controller which switch has to be modified and the parameters of the flow, in this case name of the flow, priority, on which port the packets arrive and if the flow is active. The action dictates what is done with the arriving packets, in this case they are forwarded to port 3. The HTTP request is then forwarded to the Static Flow Entry Pusher.

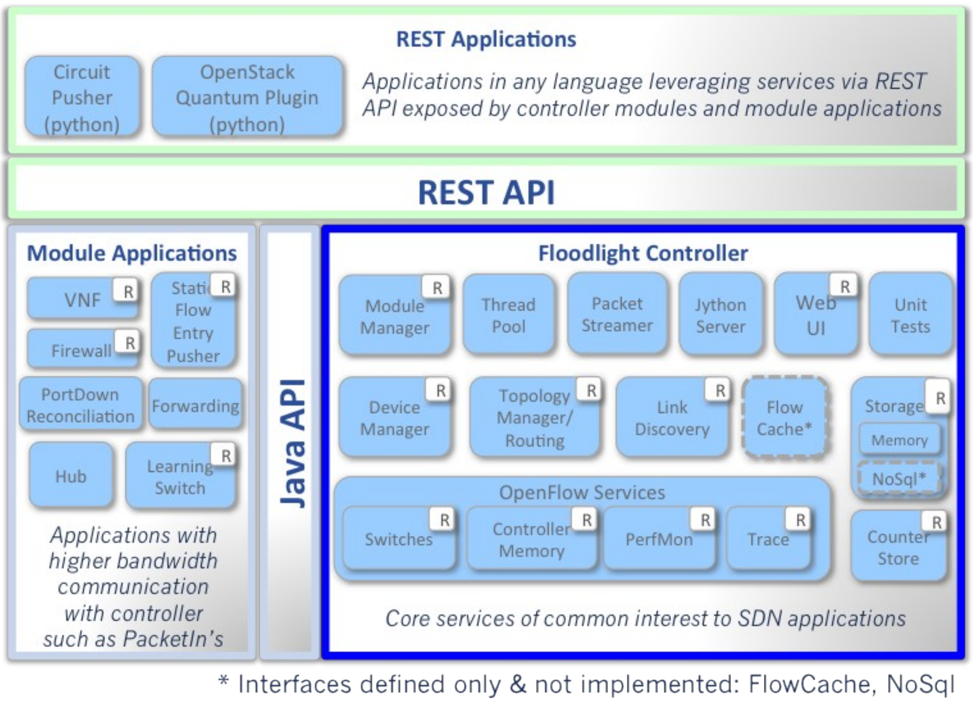

Figure [4]: The Floodlight architecture

Source: http://docs.projectfloodlight.org/display/floodlightcontroller/The+Controller

5.2 Implementation

In fact there is no real implementation. As shown above cURL can be used to access the Floodlight controller and to setup or modify flows. This command lines can be scribed into a bash-script, which can be executed. This is an easy way to test the algorithm, but on the downside it is not possible to update one switch and to wait for its reply. Therefore just a modified version of the algorithm was tested. In this modified version the controller is sending the updates to the switches in the correct order, but it is not ensured by replies that they are executed in the correct order. However tests showed that in the testbed, that was used, the switches are executing the updates most likely in the correct order. Even if this might not be the case in general, for the testbed, that was used, and therefore for the experiments this modification was no problem.

6. Evaluation

In order to test the algorithm an experiment was run. In this chapter the experiment is introduced and the results are shown and discussed. At first the topology, the goal and the results of the experiment are shown. After that the results of the experiment are discussed.

6.1 Experiment

For this experiment the topology shown in figure [5] is used, where the red links indicate the old route and the green links are the ones used by the new route. At first, only the old route is present, then a file transfer from VM 1 to VM 5 is started. The time this file transfer takes is measured. This step is executed 10 times to get more reliable results. In addition the same procedure is done with the new route (the route after the update).
After that again a file transfer is started. During this file transfer the route update is executed. Again the time required for the file transfer is measured.
Three different update strategies are tested and each strategy is tested 10 times. The first strategy is the one shown in this work, the second one is updating the switches in the same order as the packets pass them. This is the opposite direction compared the strategy described in this work and leads to black holes. The fourth strategy is a two phase update where the new route is created before the update starts, so that the update is only updating the ingress switch.
The File transfer is done by using netcat on both hosts. Netcat sets up a TCP-Connection between this two hosts. The time is measured by using the unix “time” function, which measures the time needed for executing a function. The following code was executed:

```
On VM 5: nc -l 2000 > /dev/null
On VM 1: time nc VM1 2000 < test
```

The first command makes netcat listening on port 2000 for incoming connections, if a connection is set up, the received data is dropped, because it is not needed. The second command measures the time that netcat needs to set up a connection to VM 1 and to send the “test” file, which is a 1GB big file filled with zeros.
After each single test all flow-table-entries on all switches are deleted and the old route is set up again, so that each test has the same preconditions. The updates are done by using bash-scripts for the Static Flow Table Pusher.
The goal of the experiment is to find out what influence the algorithm described in this work has on a TCP-Connection and how large this influence is compared to other strategies.
The results are shown in the table below.

	Old route without update	New route without update	End to Begin	Begin to End	Two Phase
Test 1	8.947 s	8.957 s	8.952 s	9.000 s	8.957 s
Test 2	8.949 s	8.951 s	8.953 s	8.987 s	8.951 s
Test 3	8.950 s	8.953 s	8.950 s	8.999 s	8.953 s
Test 4	8.947 s	8.955 s	8.946 s	9.775 s	8.955 s
Test 5	9.228 s	8.950 s	8.955 s	9.315 s	8.950 s
Test 6	8.956 s	8.949 s	8.951 s	9.631 s	8.949 s
Test 7	8.954 s	8.950 s	8.950 s	9.026 s	8.950 s
Test 8	8.947 s	8.951 s	8.949 s	9.905 s	8.951 s
Test 9	8.948 s	8.959 s	8.947 s	9.409 s	8.959 s
Test 10	8.947 s	8.965 s	8.950 s	9.170 s	8.965 s
min	8.947 s	8.949 s	8.946 s	8.987 s	8.949 s
max	9.228 s	8.965 s	8.955 s	9.905 s	8.965 s
average	8.977 s	8.954 s	8.950 s	9.322 s	8.954 s
Average without min and max	8.949 s	8.953 s	8.950 s	9.291 s	8.953 s

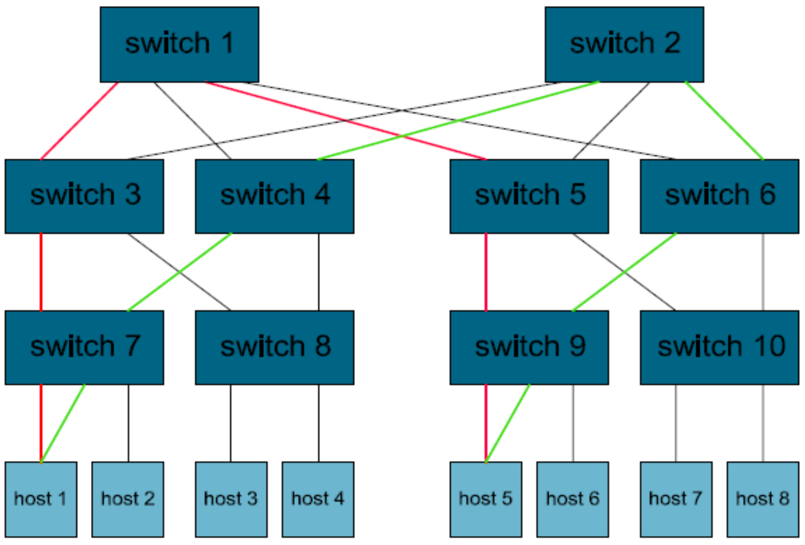

Figure [5]: The topology used for the experiment. Old route: red links, new route: green links

6.2 Discussion

In this section the results of the experiments above are discussed. The table shows the following results:
In the second column the results for the old route without update are shown. In the third column the results for the new route without update are shown. The columns four, five and six are show the results for the three update strategies. Column four shows the strategy that starts with the last switch and updates all other switches in reverse order (the approach shown in this work). Column five shows the counterpart, the update starts with the first switch and ends with the last one. Column six finally is the two phase update.
First of all it is important to notice, that both routes, the new and the old one, contain the same amount of switches, therefore the path length should be almost the same. If the outliers, the minimal and the maximal value, are removed and the remaining 8 values are compared this is indeed the case.
The next interesting fact is that the “reverse update” is much better than the update that starts with the first switch. This is the case, because the second update strategy leads to black holes, which means that packets are lost during the update, this leads to retransmissions, which further lets TCP assume a congestion, which in result slows the data transfer down.
Another interesting result is that the “reverse update” is slightly better than the two phase update, which probably is the case because the reverse update strategy needs no vlan tagging. But it is more likely that this result is based on errors in the measurements. For example the measurements for both algorithms where done at different points in time. That means the conditions within the testbed may have changed. The big outliers in the second column show that the conditions within the testbed are unstable. But both approaches lead to very similar results and are better than the update strategy that is updating the switches along the route.
Also it is interesting how small the effect of the “reverse update” and the two phase update is on the performance. According to this result it seems as the “reverse update is, at least, competitive to the two phase update strategy. But this could be different in topologies in which the reverse update strategy leads to transient loops.
To find out which of the two protocols is better under which conditions needs more tests with more different topologies.

7. Conclusion

This work shows a new approach for consistent route updates in SDN. Even if the algorithm is not leading to strict consistency, at least eventual consistency is reached.

This is reached by updating the switches on a new route, that replaces the old one, in reverse order. Reverse order means that the update starts with updating the last switch on the new route and then the one before and so on until the first switch (ingress switch) is reached.

Many other existing approaches are limitted in some ways, for example only work with specific protocols like BGP (Border Gateway Protocol), even the approch by Reitblatt et al., wich can be seen as "state of the art", suffers under the restriction that it is necessary to modify the header fields. This is necessary, because this approach is a two phase update and therefore the packets must hold the state of the phase. For example vlan tagging can be used to do that.
This algorithm on the other hand has no such restrictions, this is reflected in the system model which makes only very general assumptions. On the downside the algorithm only reaches eventual consistency, because there exist some topologies in which the algorithm creates transient loops, loops that exist during the update is executed, but not after the update is finished.
There are two possible ways to avoid this transient loops. One would be to use a controller that recognizes such topologies, which is not a big deal, because this topologies all have in comon that there are two or more switches, which are used in both routes, the old and the new one, but in different directions.
The other possibility would be to modity the algorithm which is beond the scope of this work, but would most certenly be no big deal.

The experiment that was run in this work furthermore shows, that the algorithm is, at least in the topology that was used for the experiment, competitive to a two phase protocol in terms of performance. Also the ordering, that the algorithm uses to update the switches, achives much better results than a different ordering that updates the switches in the same direction as packets would pass them, which is the opposite direction compared to the one that the algorithm, shown in this work, uses. Probably the most surprising result of the experiment was, that the update algorithm has a very sligtly effect on the performance of a TCP file transfer that is executed before, during and after the update.

Further work is required to test the algorithm in more different topologies, for examples in topologies for which the algorithm creates transient loops. Furthermore the effect on a UDP file transfer could be tested and compared to the results for TCP-Connections. Also the algortihm could be modified in a way that it can avoid transient loops in certain topologies.

8. Bibliography

[1] Open Networking Foundation. Software-defined Networking: The New Norm for Networks, Apr. 2012.

[2] Open Networking Foundation. OpenFlow switch specification, June 2012.

[3] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, pages 323-334, Helsinki, Finland, Aug. 2012.

[4] R. McGeer. A safe, efficient update protocol for OpenFlow networks. In Proceedings of the First Workshop on Hot Topics in Software-defined Networks, page 61-66, Helsinki, Finland, Aug. 2012.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Perterson, J. Rexford, S. Shenker, and J.Turner. OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69-74, Apr. 2008.

Erklärung

Ich Versichere , diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines Anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder vollständig noch teilweise veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

(Stuttgart, 13.05.2013)