Till Kleisli, tillk@student.ethz.ch

Critical Evaluation of *(e)POST* and *Porcupine*

What is a Distributed Mail Server?
Even if both papers have the same topic, the resulting system is quite different. I didn't find any explicit statement about it, in both papers, but from details you can guess that the two groups of authors had something different in mind, when they wrote their paper. *(e)POST* is rather designed to be used on desktop PCs that are also used as normal workstations. Unlike *Porcupine*, where such a usage wouldn't make sense for different reasons. The effort to rebuild the whole index structures for example, if a node leaves or joins the cluster, meaning turning off or on a personal computer. For *Porcupine* it makes more sense to use it on a dedicated cluster of small machines, like google. The problems with *(e)POST* on a dedicated cluster have different reasons, one of it is its bad interoperability with the existing email infrastructure.

The Principle of Functional Homogeneity
The principle of functional homogeneity, meaning that every node can perform any function, is used in both systems, even though only *Porcupine* states it explicitly. I think this is a principle principle for distributed systems, because the functionality of a system should also remain available, even if some nodes fail. You could also implement a quorum-based solution, like in data replication, where a quorum of nodes implements a given functionality. But I think that isn't really needed, because functionality normally doesn't change over time (at least not very often), so you don't get update problems. And functionality normally also doesn't need much disk space, so it can easily be replicated at all nodes.

Secure Email
Porcupine handles emails as today's mail servers like postcards. Anyone who has access to the post office or the mail server can read any mail. So it doesn't get worse, but it also doesn't get better. A good point on *(e)POST* is the built in digital encryption and singing of messages. That means, functionality that has to be added manually nowadays by installing PGP for example, is already built into the mail system. Which is needed, because in *(e)POST,* every user potentially stores other users mail on his desktop and a spy wouldn't even have to get access to the mail server any more, to read other users mail. But on the other hand this functionality needs a certificate authority which is not included in the *POST* system, so you also have to combine it with other software, to use it.

Disk Space
We don't have to consider disk space in the *Porcupine* system, because in principle its like a normal mail server, just distributed. So if data can be handled on one server it can also be handled on a distributed system.
In *(e)POST*, there are two „special features“ concerning disk space. The positive one is possible, because the headers and the bodies and attachments of emails are stored separately. So if an attachment is sent by many people, or a message is forwarded, the attachment or the message body has to be stored only once, and the header contains a reference to the body or the attachment respectively. This is a nice solution to save disk space, but *(e)POST* needs such solutions, because in the underlying storing system *PAST*, no objects can be deleted. That means every mail that has ever been sent or received through the *(e)POST* system is stored forever in *PAST*. In the *POST* paper they state: „Thus, the amount of available disk space in the system must be increasing and greater than the total storage requirements, which is reasonable to expect in a p2p environment where each participant is required to contribute a portion of her desk top's local disk.“. I actually don't see why this should be reasonable, and it reminds me of a economical principle called „Snowball system“, which only works if the number of participants is increasing exponentially. And which is forbidden in Switzerland, by the way.

Performance
There is no part dealing with the performance of the *(e)POST* system in the paper. Probably they don't have any data because there exist prototype implementations of *ePOST* and *POST*, but they are currently under experimental evaluation.
As opposed to the *Porcupine* paper, where almost half of the paper is about System Evaluation and Performance. They built a heterogeneous cluster with 30 nodes as a prototype implementation. The assumed model is described in detail in the paper, but I will just summarize the results here. With no replication, *Porcupine* can handle much more messages per second than a conventional mail server, but that doesn't make much sense. For replication, the performance of *Porcupine* scales linearly when each incoming message is replicated on two nodes. With a higher replication factor, the performance would fall even deeper, but I think that's a reasonable price for availability and resilience. The performance can be improved by using tricks like non-volatile RAM or multiple disks per node.

Usefulness
As I stated in the first paragraph, the two systems probably didn't have the same goal, but nevertheless, I would prefer *Porcupine* as my distributed mail system. Mainly for the reason, that it interoperates better with the existing infrastructure. *(e)POST* is rather designed to build a small world by itself, and you have to have gateways to all the other mail users to which you may want to send mail or from which you want to be able to receive mail.
This doesn't mean *POST* is useless. I just think *ePOST* is not the best case to use the *POST* system, although you could use it as an internal mail system for a company. But I think its more convenient for intra-company instant messaging, shared calendars for specific user groups or other applications that don't need any gateway to outside the company. Instead of a company you could assume any kind of closed community.