universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

„Quantum Cryptography in the Tokyo Metropolitian Area“

Verfasser

Andreas Allacher

angestrebter akademischer Grad

Magister der Naturwissenschaften (Mag.rer.nat.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 411
Studienrichtung lt. Studienblatt:	Diplomstudium Physik
Betreuer:	o.Univ.-Prof. Dr. DDr.h.c. Anton Zeilinger

Work supported by the Austrian Science Fund (FWF), Project SFB-FoQuS F 4008-N16.

Contents

1.). Abstract

This work presents an entanglement based quantum key distribution (QKD) system for use in optical telecommunication fibres.
It uses two modules Alice and Bob and one QRNG for e.g. privacy amplification.

The Alice module also contains the entanglement source and measures the 810nm photons. The Bob module measures the 1550nm photons of the entanglement source after they pass through the optical telecommunication fibre.

Furthermore, this work shows how to achieve stablized QKD through fibres by using various software automated stablization and optimization procedures (better explained in chapter 6.)).

It also describes how the system counteracts new found side channel attacks (Makarov), see chapter 8.).

It also shows how to integrate the system in a QKD network with other quantum cryptography systems during the Tokyo QKD network demonstration (at UQCC 2010) and that it achieves a stable keyrate and QBER during that demonstration.

The network itself and other quantum cryptography systems used in the
Tokyo QKD network demonstration are outlined in the paper
”Field test of quantum key distribution in the Tokyo QKD Network”, Optics Express, Vol. 19, Issue 11, pp.10387-10409 (2011) [1].

2.). Quantum cryptography

2.1. Classical Cryptography

Cryptography is a method to deliver messages between two parties that will result in these messages being unreadable by a third party if intercepted. It is not necessary that such messages are unreadable (to a third party) for eternity but it is enough for them to be unreadable as long as they are valuable.
Nowadays there are a few more applications like authentication, signatures, etc. (further details can be found in [2]).

Cryptography works by using an algorithm which combines the message with a key. This method is called encryption and produces an encrypted message. In the best scenario it should be impossible to read the message without the correct key, however, this is often weakened to just being difficult without the key.

There are two main algorithm types that are used to encrypt messages: Symmetric encryption (for these there is a special case called one-time pad encryption) and asymmetric encryption.

2.1.1. Symmetric encryption

Symmetric encryption algorithm use the same key for encrypting from and decrypting back to the original message.
The messages are encrypted by using keys. These encrypted messages are then sent to the receiver who already knows the key and then decrypts the messages.

This method is more secure than asymmetric encryption schemes, however, it has one big drawback and that is how to distribute the key without being interecepted.
Normally the key length for symmetric encryption is way lower than the message length itself. In these cases the message itself is divided into blocks and combined with the key through complex algorithms.

This encryption (where the key-length is lower than the message itself) is only computational secure but it is still more secure than asymmetric encryption systems.
However, the problem with symmetric encryption systems is how to securely distribute the key. If the key would be intercepted during transport, the whole message could be decrypted.

That is why keys are either transported by a trusted person or by using asymmetric encryption

algorithms or something similar.
The reason for not using asymmetric encryption for whole messages is because they are slower than symmetric algorithms.

A special case of of symmetric encryption schemes is

2.1.2. one-time pad (OTP)

It is a method where the key has the same length as the message. It just adds each bit of the of the key to the corresponding bit of the message. Then the encrypted message is sent to Bob who subtracts the key from the message and the result is the original message.

For this method to be secure it is important for the key to be completely secure which can only be achieved with a completely random key. If that is the case then this encryption method has one huge advantage to all the other encryption schemes because it is proven that such a scheme is secure according to information theory (proof by Shannon in 1949).

However, it has still the same drawback as other symmetric encryption schemes and that is to distribute the key itself without losing security. It is actually worse as the key is as long as the message and if we want it to be proven secure we can't use asymmetric encryption.
The only way in classical cryptography would be a trusted person that delivers the key. However, quantum cryptography provides a solution to this problem (see chapter 2.2).

2.1.3. Asymmetric encryption

Asymmetric encryption methods are also known as public-key encryption methods.
The main difference between symmetric encryption methods and these is that for asymmetric encryption methods we have two types of keys. The first type is the so called public key and the other is the private key (see also [2]).

In those algorithms the receiver (called Bob) most first choose a private key. From this key a public key is computed. This public key can be distributed to anyone without weakening the security of asymmetric encryption methods.
However, the private key must NOT be disclosed to anyone for these cryptosystems to be secure.

After Bob has both the private and the public key, the public key is then given to anyone who is interested in sending Bob a secure message which only Bob and the sender (called Alice) are capable of reading.
Alice takes the public key to encrypt a message and sends the message to Bob. Bob then uses the private key to decrypt the message (the public key can NOT decrypt the message; only the private key can) and by doing so the message was successfully and securely delivered.

The main drawback is that this system is based on computational complexity. However, even more so than the symmetric counterparts because they don't just divide the message into blocks (which only is necessary if the key length is lower than the message length, of course) but the actually main "issue" is to that it is based functions which are easy to compute in one direction (meaning if I have a value "x" it is easy to calculate "f(x)") but on the other hand it is difficult to reverse the process (meaning to calculate "x" from "f(x)"), see also [2].
Which means that I can easily calculate the public key from the private key but not the other way around. That would take way longer.
The time to calculate the private key from the public key grows exponential, whereas it only grows polynominal to calculate the public key from the private key.

Most of the current asymmetric cryptosystems rely on factorization to achieve this, however, it isn't actually proven that factoring is indeed "difficult" to calculate. There is just no current algorithm whhich can do so easily in classical computation.
However, there is already an algorithm that would make this calculation easy by using quantum computation. But quantum computation is still in its research stage and doesn't yet provide enough capabilities to perform factoring for currently used keys (further information regarding this can be found in e.g. [3] and [4]).

2.2. Quantum Cryptography

Quantum cryptography is actually not related to the process of encrypting messages. In most cases OTP is used for this purpose. It is also possible to use other conventional symmetric encryption systems, however, by doing so we loose the advantage of the security proof.
So it is best to use OTP in combination with quantum cryptography.

Quantum cryptography itself is used to distribute the key between Alice (sender) and Bob (receiver) without someone being able to intercept it. At least not without us noticing which would result in us discarding those bits of the key that got intercepted. Therefore, it is also known as quantum key distribution (QKD).

2.2.1. Quantum Information

Before explaining QKD it is necessary to explain a bit about quantum information as various principles of quantum information are necessary for QKD.

Qubit

Qubit is commonly used in quantum information. It takes advantage of superposition. It uses two states that are orthogonal to each other. Those two states build a basis.

In case of polarization - which our system uses - these can be e.g. $|H\rangle$ and $|V\rangle$.
A Qubit can be any superposition between those two states like this:

$$|\psi\rangle = \alpha\,|H\rangle + \beta\,|V\rangle \text{ with } |\alpha|^2 + |\beta|^2 = 1$$

There are two more bases that are relevant for our quantum cryptography system. One of them is only necessary to control the polarization during the QKD operation (because our system requires well aligned polarisation - especially the orthogonality between the states of a basis is important).
In total two basis are used for QKD itself.
These basis are $|H\rangle$, $|V\rangle$ and $|P\rangle$, $|M\rangle$. Those basis are used for polarisation control too.
The additional basis that is only used to control the polarization is $|L\rangle$, $|R\rangle$.

The used states are defined like this:

$\|H\rangle$	horizontal (90°)
$\|V\rangle$	vertical (0°)
$\|P\rangle = \frac{1}{\sqrt{2}}(\|H\rangle + \|V\rangle)$	+45° linear
$\|M\rangle = \frac{1}{\sqrt{2}}(\|H\rangle - \|V\rangle)$	-45° linear
$\|L\rangle = \frac{1}{\sqrt{2}}(\|H\rangle + i\,\|V\rangle)$	left-handed circular
$\|R\rangle = \frac{1}{\sqrt{2}}(\|H\rangle - i\,\|V\rangle)$	right-handed circular

Qubit measurement

Here we have an important property of quantum information which is required for quantum cryptography. That is that a Qubit cannot be measured without destroying its superposition.
If we combine this with the ability to use two different basis (at random) for measurement which aren't orthogonal to each other (for instance the $|H\rangle$, $|V\rangle$ basis and the $|P\rangle$, $|M\rangle$ basis) we get security for quantum cryptography.
This is because I cannot determine the state of a qubit (through measurement) with certainty without knowing the preparation base.

As an example if we look at the probabilities to measure state $|P\rangle$ in the $|H\rangle$, $|V\rangle$ basis we get the probability of 50% that it would be $|H\rangle$ but also with the same probability it could be $|V\rangle$. However, if it is measured in the $|P\rangle$, $|M\rangle$ basis it will always be $|P\rangle$.

No-cloning theorem

This theorem is important for the security of quantum cryptography because of this theorem it is possible to completely prevent man-in-the-middle attacks. Man-in-the-middle attacks are attacks where there is an adversary (let us call it Eve) which stands between the sender (Alice) and receiver (Bob). Eve then intercepts messages send from Alice, reads it, creates a perfect copy and forwards it to Alice. In this case nobody would notice that Eve now also has this information. This way if we distribute a key, Eve would have the key and could then read all secure messages that use that key because Eve is capable of creating a perfect copy.
However, in quantum cryptography it is impossible to create a perfect copy of a Qubit and therefore this attack won't work. The reason is that we would notice it and could discard the intercepted key.
The no-cloning theorem was first shown by Wooters and Zurek (check out [20]).
In order to show the no-cloning theorem we just have to use linearity and superposition of quantum mechanics.

Let's say there exists a general cloning machine which copies a state onto a blank state $|b\rangle$.
This machine does the following:

$$|H\rangle\,|b\rangle \rightarrow |H\rangle\,|H\rangle$$
$$|V\rangle\,|b\rangle \rightarrow |V\rangle\,|V\rangle$$

In this case if we use a superposition state like $(|H\rangle + |V\rangle)$ because of linearity the result is:

$$(|H\rangle + |V\rangle)\,|b\rangle \rightarrow |H\rangle\,|H\rangle + |V\rangle\,|V\rangle$$

Whereas the result should be:

$$\begin{aligned}(|H\rangle + |V\rangle)\,|b\rangle \quad &\rightarrow \quad (|H\rangle + |V\rangle)\,(|H\rangle + |V\rangle) = \\ &= |H\rangle\,|H\rangle + |H\rangle\,|V\rangle + |V\rangle\,|H\rangle + |V\rangle\,|V\rangle\end{aligned}$$

Therefore the result and the should-be result differ which means that there exists no cloning machine that could copy any arbitrary state.

2.2.2. Quantum key distribution

This is the main part about quantum cryptography. It is the possibility to distribute a key without anyone being able to intercept it because once they do our exchanged key size is reduced to zero bits.
If we combine this distribution method (QKD) with OTP we get a perfectly secure system. It is even theoretically proven that QKD is unconditionally secure. At least most of the systems are unconditionally secure. Most systems (like our system) are based on polarization and photons but there are new systems that use differential phase shift (DPS) which is not yet proven to be unconditional secure but just for certain scenarios (further information see [1]).

Unconditional security also means that even if an adversary (so-called Eve) has better equipment than ourselves (even if it is the best theoretically possible equipment) the key distribution between Alice and Bob is still completely secure.

QKD takes advantage of the no-cloning theorem (see chapter 2.2.1) and that without knowing the preparation base it is not possible to determine the state with certainty (see chapter 2.2.1). With these properties it achieves a secure distribution method for the keys. In principal there are many protocols for quantum cryptography that take advantage of these properties to create a secure distribution method.
The oldest protocol is the so-called:

BB84

This protocol (proposed by Charles H. Bennett of IBM and Gilles Brassard of the University of Montreal in 1984) uses two bases for quantum cryptography, e.g. like our system ($|H\rangle, |V\rangle$) and ($|P\rangle, |M\rangle$).
So there exists four states and we have to assign binary system values to them (each base should have a 1 and 0), so for instance $|H\rangle$ and $|P\rangle$ are assigned to "0" and $|V\rangle$ and $|M\rangle$ are assigned to "1".
Now Alice chooses randomly one of those four states and sends it to Bob.
Of course, how to choose a state really random is not that easy itself but there exists various random number generators that are - at least in theory (in praxis there might be limitations due to imperfect equipment) - producing completely random numbers (like a "Quantum Random Number Generator (QRNG)", see chapter 3.2.5 for more details).
This state is then sent to Bob and Bob chooses one of the two bases randomly (again really random but independent from Alice) to measure the state.
If Alice and Bob used the same basis the get correlated results. However, if they don't they get uncorrelated results.
Because of this method the error rate of the received "key" has an average of 25 % which is too high for normal error correction methods.

This is why we need to do the so-called:

2.2.3. Sifting

This process gets rid of uncorrelated results because Bob used the wrong basis for measurement. Bob sends the used measurement basis to Alice through a public channel (everyone is allowed to obtain this information, it doesn't interfere with the security of QKD) and Alice only answers if its measurement basis was the same or not.
Though as said it can be a public channel it should be authenticated. This means that it should be verified that the messages are from Alice or Bob and not from an adversary Eve. In most cases this is achieved by using key material that has already been distributed. This key material can be key material that had already been exchanged through QKD or if the system was just installed it would be pre-distributed key material, e.g. by copying the material through a data storage medium.
If Alice used the same measurement basis as Bob the bit is kept, otherwise it is discarded. That way we loose about 50 % of the received key.
By using this method neither Alice nor Bob alone decide which key results from the protocol. The resulting key is produced by using random choices on both sites.

If Eve would now intercept the transmission the bit-rate of the key would go down (it could possibly even become zero). This happens because Bob would have expected a bit but didn't receive it and therefore just needs to tell Alice to discard that bit.

However, Eve would try to create a copy of the bit and forward it to Bob in order to stay hidden from the system.
This is the so-called:

2.2.4. Intercept-Resend attack

The intercept-resend attack means that Eve creates a copy of the measured state and sends it to Bob. However, Eve doesn't know in which basis Alice sent the qubit, she has to choose one of the two basis for measurement.
She then sends to Bob a state according to her measurement. However, in about half of the cases Eve chooses the wrong basis and therefore forwards the wrong state to Bob. In the other half of the cases she sends the correct states to Bob in which Alice and Bob couldn't detect Eve's presence. If we then take all the received material at Bob after sifting (because Bob also has to choose a measurement basis) we would get an error rate of approximately 25 % which is high enough to detect. Therefore, we know that Eve tries to get our key material and we can discard the whole material.
Of course, there is also the possibility that Eve only tries to get a part of the communication. This would result in lower error rates but she would get parts of the key material.
Henceforth, for quantum cryptography further steps are required to ensure perfect security

(mainly privacy amplification). However, before privacy amplification we first have to get rid of errors resulting from technical imperfections, intervention by Eve etc.

2.2.5. Error-Correction and privacy amplificiation

Error correction uses classical error correction algorithms to reduce the error rate caused by technical imperfections or by an adversary Eve from a few percent (typical in current QKD systems) to the typical error rate of classical optical communications.
This is done after sifting and the error rate is called QBER (quantum bit error rate).

The simplest method for error correction would be to randomly create pairs of bits and create XOR values and announce those XOR values publicely at Alice. Bob then replies if the XOR value matches or not.

If the XOR value matches both systems keep the first bit of the pair and discard the second one. On the other hand if the XOR value does not match then both bits are discarded.
Of course, in reality more efficient and complexer algorithms are used (like e.g. CASCADE - for further information see [23] - or LDPC - for further information check out [25]).
After this error correction both (Alice and bob) have the same key. However, Eve might still have some information about it and therefore the last step is privacy amplification.

Privacy amplification (further information than in this chapter can be found in [21]) is used to reduce Eve's information on the final key to a minimum.
One simple privacy amplification method would be for Alice to again choose randomly pairs of bits and calculate their XOR value. However, this time she only announces which bits she chose but not the XOR value. Afterwards, Alice and Bob replace those two bits with the calculated XOR value. This way the key gets shorter but stays error free and if Eve only knows one bit she knows nothing about the XOR value. Or if Eve only knows each bit with a certain probability then the probability of the XOR value is even less.

By repeating this process it would be possible to get the information Eve has down to an arbitrary value. Of course, there are better algorithms to achieve this.
Furthermore Norbert Lütkenhaus described in [22] a bound for the maximum information Eve can obtain for a certain QBER and by using this we can indeed reduce Eve's information not just to an arbitrary value but to a minimum.

2.2.6. Ekert protocol

This protocol (proposed by Artur Ekert in 1991) requires entanglement (described in chapter 3.1). As our system uses entanglement this protocol should also be mentioned although it isn't used by our system.
In this protocol instead of sending qubits from Alice to Bob, a common source is used which sends qubits to Alice and to Bob.

This can be achieved if the qubit sent to Alice and the qubit send to Bob by the source have the same state. This state is chosen randomly. The source then announces the basis.
If Alice and Bob used both the same basis as the source they keep the data.
Equivalent to the BB84 protocol this results in about half of the cases using the correct basis.
If the source is reliable this protocol is equivalent to the BB84 protocol.

However, Ekert proposed that instead of trusting the source it would be better to use an entanglement source with a maximally entangled state like:

$$\left|\psi^{-}\right\rangle = \frac{1}{\sqrt{2}}\left(|H\rangle \otimes |V\rangle - |V\rangle \otimes |H\rangle\right)$$

In this case if both Alice and Bob use the same measurement basis (they don't require any information from the source) their results are directly inverse to each other. This means that we would e.g. have $|V\rangle$ at Alice and $|H\rangle$ at Bob. Therefore, we just have to assign corresponding bit values at Alice and Bob. However, instead of previously where we used the same bit for the same states at Alice and Bob, we now have to swap them at one site - e.g. at Alice $|H\rangle$ is "0" then at Bob it is "1" and therefore $|V\rangle$ at Alice is "1" and at Bob "0"). This way we get the same bit as result at Alice and Bob.
Of course, for certain different maximum entangled state it is possible to use the use the same bit for the same state at Alice and Bob.
By using this method the choice is again correct for about half of the cases (which is similar to BB84).

Furthermore, Ekert suggested to base the security of this protocol on the Bell inequalities (see chapter 3.1). In order to do so Alice and Bob would use a third basis. Therefore, to check the Bell inequalities and to have the same measurement bases, only about $\frac{2}{9}$ of the raw material remains after sifting.

The advantage for checking the Bell inequalities is that we can ensure that the source really emits entangled states.

2.2.7. BBM92

This protocol (proposed by C.H. Bennett, G. Brassard, N.D. Mermin in 1992, check out [26] for more information) is basically the same as BB84 but instead of Alice choosing a random state, it uses an entanglement source at Alice.
There Alice creates a pair of maximally entangled particles (also check out chapter 3.1) and measures one of them at a random basis. This results in one of the four used states by the system and as required this resulting state is randomly chosen. The other particle is sent to Bob which is therefore also in a random (now known - because we measured one of the particles) state.
Therefore, it is also similar to the Ekert protocol, however, it doesn't require the testing of Bell inequalities.
It would also be possible to use the entanglement source at a different location from Alice and it would still be secure without testing the Bell inequalities.

3.). Our system

3.1. Entanglement

As our QKD system uses an entanglement source it is necessary to explain entanglement to understand how it works.
Furthermore, the whole system uses the BBM92 protocol which is described in chapter 2.2.7 with the entanglement source being at Alice.

For pure states there exists pure states which can be written like: $|\phi\rangle_{AB} = |\phi\rangle_A \otimes |\phi\rangle_B$

Such states are product states and are seperable, however, not all states are like this.
The most general state $|\phi\rangle_{AB} = \sum_{i,j} c_i j \, |\imath\rangle_A \otimes |\jmath\rangle_B$
is only seperable if $c_i j = c_i^A c_j^B$. Therefore, if $c_i j \neq c_i^A c_j^B$ it is not seperable and such states are called entangled states.

The bell states are the maximally entangled states of two Qubits:

$$|\psi^-\rangle = \frac{1}{\sqrt{2}}\left(|H\rangle_A \otimes |V\rangle_B - |V\rangle_A \otimes |H\rangle_B\right)$$

$$|\psi^+\rangle = \frac{1}{\sqrt{2}}\left(|H\rangle_A \otimes |V\rangle_B + |V\rangle_A \otimes |H\rangle_B\right)$$

$$|\phi^-\rangle = \frac{1}{\sqrt{2}}\left(|H\rangle_A \otimes |H\rangle_B - |V\rangle_A \otimes |V\rangle_B\right)$$

$$|\phi^+\rangle = \frac{1}{\sqrt{2}}\left(|H\rangle_A \otimes |H\rangle_B + |V\rangle_A \otimes |V\rangle_B\right)$$

The $|\psi^-\rangle$ state is used in most cases in quantum cryptography for transport because it has rotational symmetry.
Therefore, the state has the same form in a different measurement basis. This is convenient as quantum cryptography requires multiple measurement bases to be secure.

3.2. QKD System

There is an Alice and a Bob side of our system.
All components at one side are stored inside a 19” rack and can therefore be viewed as units.
Our entanglement source is included in Alice. Therefore it utilises the BBM92 protocol for quantum entanglement as the entanglement source and Alice are at the same point instead of the ”Ekert protocol” which requires a third component as entanglement source.

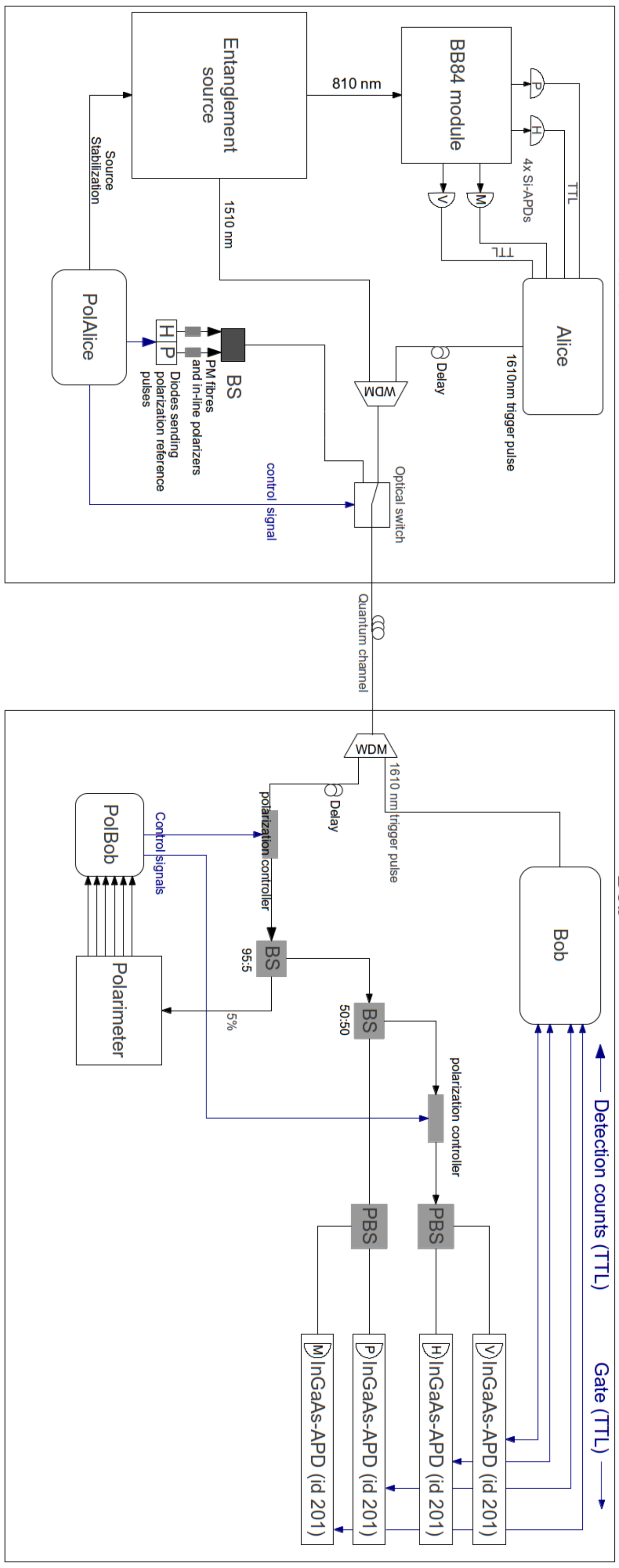

Figure 3.1.: Layout of the entanglement-based QKD system used in UQCC2010.

3.2.1. Alice

Alice contains the polarization-entanglement source which is described in ”High-fidelity transmission of polarization encoded qubits from an entangled source over 100 km of fiber” [13].
.

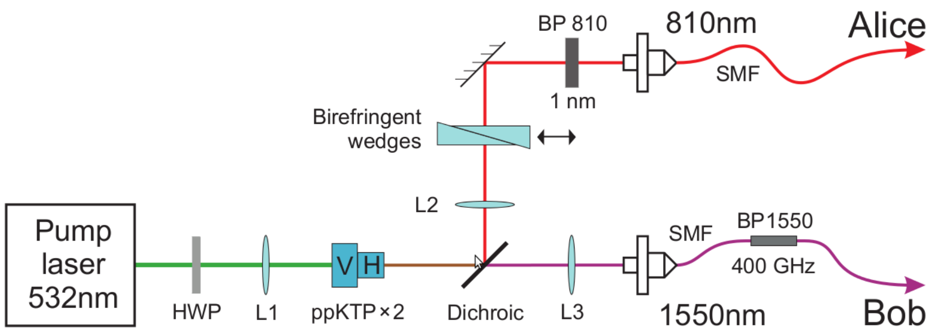

Figure 3.2.: Entanglement source for QKD system, from ”High-fidelity transmission of polarization encoded qubits from an entangled source over 100 km of fiber” [13, Page 7856 Fig. 2]

”The two nonlinear crystals used in the source are quasi-phase matched periodically-poled KTiOPO$_4$ (ppKTP), with a grating spacing of 9.7 μm, which has been tailored for type-I collinear generation of an asymmetric photon pair at 810 and 1550 nm from a 532 nm pump.” (cite from [13, Page 7856]).

There are down-conversion processes in the two crystals. This way the source generates two photon pairs which are in- distinguishable in terms of spectral, spatial and temporal degrees of freedom and therefore the photon pair doesn’t reveal in which crystal it was produced (further information see [13, Page 7856]). These down-conversion processes produce two photon pairs which are in- distinguishable in terms of spectral, spatial and temporal degrees of freedom, the presence of a photon pair does not reveal in which crystal it was produced. This leads to this state:

$$|\phi\rangle = \frac{1}{\sqrt{2}}\left(|H_{810}\, H_{1550}\rangle + e^{i\phi}\,|V_{810}\, V_{1550}\rangle\right)$$

There is one remaining issue, however, which is that through chromatic dispersion of 810 and 1550 nm it leads to a temporal distinguishability. To get rid of this the birefringent wedges are used. Furthermore, they allow to tune the phase of the state.

After the entanglement source produces the photons the 810nm and 1550nm photons are split using a dichroic mirror into single mode fibres.
The 810nm photons then pass through an in-fibre polarization controller and enter the ”BB84

module".
This module uses a beam splitter (BS) and two polarizing beam splitters (PBS).
The PBS are differently rotated by 45° along the transmission axis to be able to measure the H/V and P/M basis.
The four outputs of the BB84 module are then coupled into four Si-APDs (SPCM-AQ4C from Perkin Elmer) and their outputs are connected to an electornic board (Xilinx Virtex 4 FX20).
The whole system uses four FPGA boards (Alice, PolAlice, Bob and PolBob) with Xilinx Virtex 4 platform and embedded CPU (IBM PPC405), further information regarding the FPGA boards can be found "A Fully Automated Quantum Cryptography System (Diploma Thesis)" [7].
Furthermore, for every photon detection at Alice a strong 1610nm laser pulse is merged with the 1550nm signal (through a WDM) and
sent to Bob to synchronize the detection events between Alice and Bob.
This is necessary to open the detector gates at Bob at the correct time and to know which photon pairs are coincidences.
The 1610nm signal is delayed by a delay fibre to ensure the signal is sent after the photon. Tis fibre might need adjusting if the distance between Alice and Bob is too long or too short because the 1610nm signal is faster than the 1550nm signal. Therefore, it could either be too far (if the delay fibre is too long) to be compensated by the software delay because of a shorter distance between Alice and Bob (see chapter 6.2) or even worse the delay signal could bypass the 1550nm photon used for QKD if the distance is too long. Furthermore, the detection event is delayed before sending the sync-pulse to prevent side channel attacks (see "A Fully Automated Quantum Cryptography System (Diploma Thesis)' on details regarding those side-channel attacks' [7]).
PolAlice is used for Source Stabilization and Polarization control between Alice and Bob.

3.2.2. Source stabilization

To provide Source stabilization the following elements of the entanglement source can be adjusted electornically (by using piezo mounts with two tilt axis):
fibre couplers
mirror after the laser

It uses the hill-climber algorithmus to achieve better photon rates.
Futher details can be found at chapter 6.2

3.2.3. Polarization control

Polarization control is done by sending strong reference pulses from Alice to Bob. Those reference pulses are coupled into PM (polarization maintaining) fibres. However, it turned out that because of coupling problems these fibres alone were not capable of keeping a stable polarization

control. Only if perfectly coupled into PM (polarization maintaining) fibres the polarization is really stable. This, however, is not the case for the used in-fibre reference laser diodes.
Therefore, after coupling we now pass in-line polarizers (one per diod) to get stable polarized reference pulses.
In order to separate those reference pulses from the normal Quantum channel photons an optical switch is used. However, on Bob's side a 95/5 beam splitter is used to direct a fraction of the light to a six-channel polarimeter.
This 5% fraction of the light is analyzed by first splitting it in three equals parts. Those are then analyzed in the two linear (H/V and P/M) and one circular (R/L) basis.
After the state is anaylized PolBob calculates the the deviation angles to the target state and applies corresponding voltages to the polarization controllers to achive optimization of the incoming state's polarization (further details regarding Polarization control can be found at chapter 6.2 and in "A Fully Automated Quantum Cryptography System (Diploma Thesis)"[7, Chapter 6.3] and regarding polarization control in general see [27], [28] and [29]).

Some pictures of Alice that were taken in the lab during the UQCC2010 preparation in Tokyo.

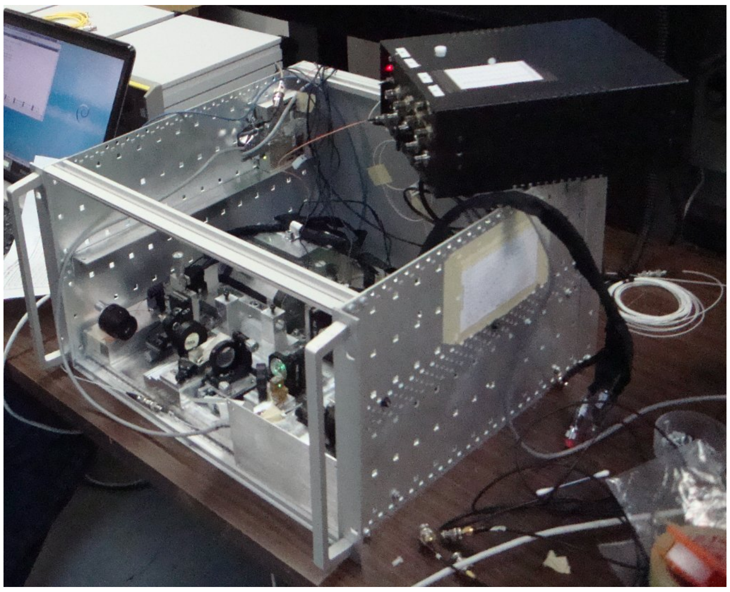
Figure 3.3.: Shows Alice with opened cover

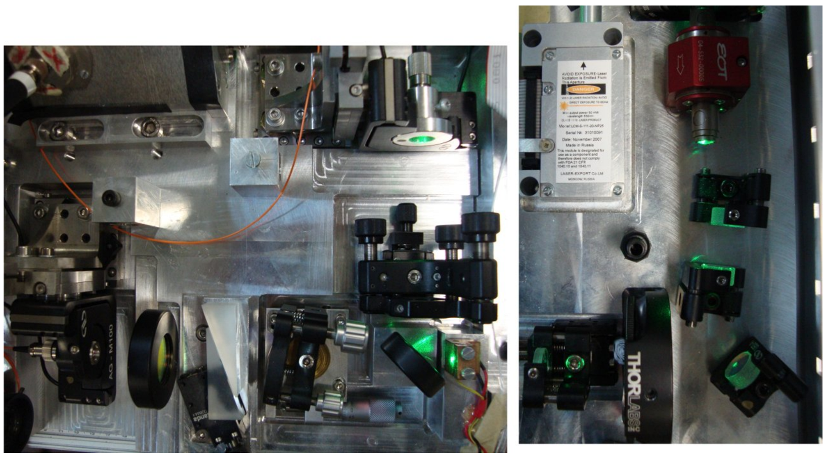
Figure 3.4.: Shows the internal parts (mainly entanglement source) of Alice

3.2.4. Bob

Bob first uses a WDM demultiplexer to split the trigger signal (1610nm) from the photons used for quantum cryptography (1550nm).
The 1610nm signal is then converted into an electric trigger signal (TTL) by a FPGA board. This is used to trigger the gates of the InGaAs single photon detectors (id Quantique id201).
The 1550nm photons pass a 32m delay fibre becaue the trigger signal arrives after the photon. Afterwards. they pass go through a polarisation controller General Photonics PolaRite II [24] (for PolCtrl see chapter 6.2). Then a 95/5 beam splitter is used to split a fragment (5 %) of the incoming light. This fragment is used to analyse the polarization control reference pulses sent by Alice, see chapter 3.2.3
Futhermore it can be used to detect and therefore prevent bright light Makarov attacks, see chapter 8.1.
The other part of the light (95 %) passes a 50/50 beam splitter because we need to analyze two bases and then one part of the beam splitter passes another polarization controller to adjust only one basis (H, V). In both arms then PBS (polarized beam splitters) are used to provide P, M, H, V outputs to be measured.
Those for outputs are normally measured by four InGaAs detectors id Quantique id201. Though in Japan we decided to use only three id Quantique id201 and one old id Quantique id200 because that certain one has better detection probability and less dark counts as our tests show.

Bob's layout is shown at the bottom part (or right side if viewed in the correct rotation) of Figure 3.1

More details regarding Bob (as it was also used in SECOQC) can be found in the diploma thesis from Alexander Treiber: "A Fully automated entanglement-based quantum cryptography system for telecom fibres" [7, chapter 5.2].

Some pictures of Bob that were taken in the lab during the UQCC2010 preparation in Tokyo.

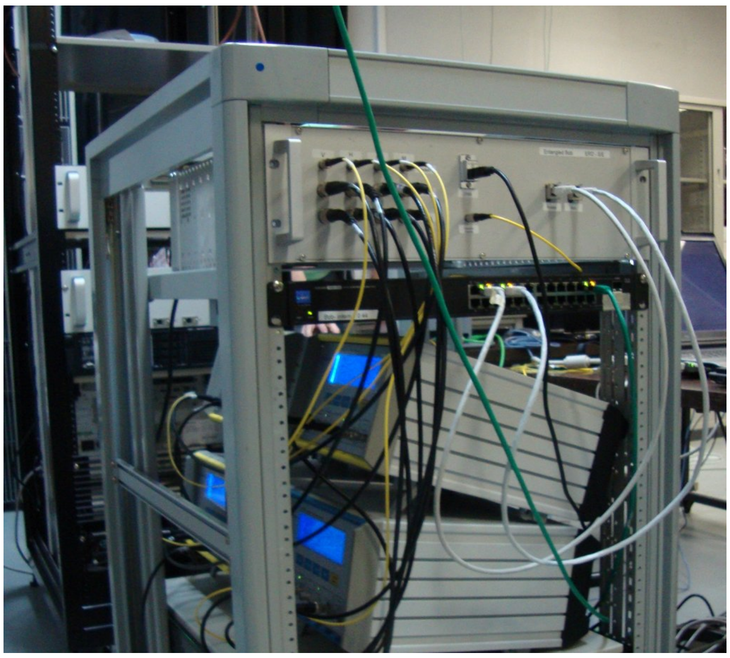

Figure 3.5.: Shows Bob with the detectors

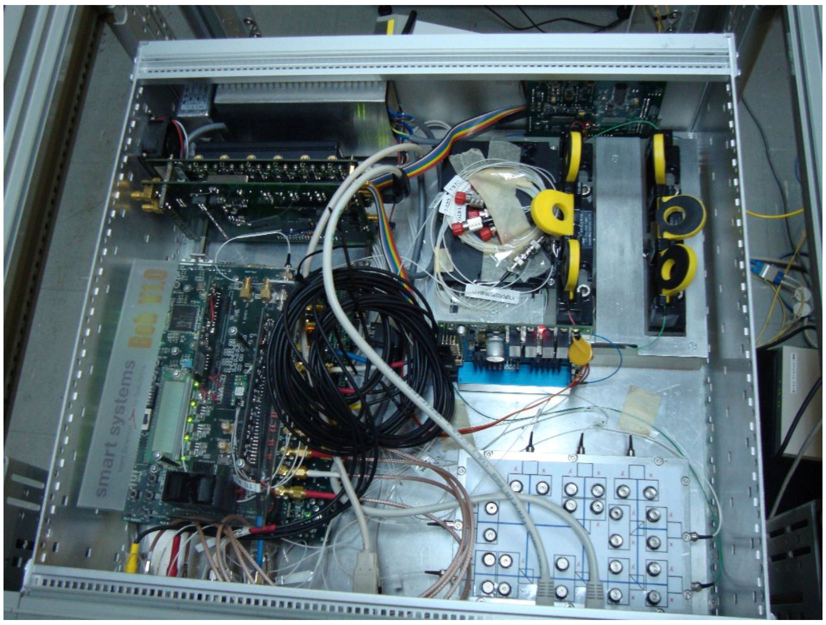

Figure 3.6.: Shows the internal parts of Bob

3.2.5. QRNG

QRNG stands for "Quantum Random Number Generator".
It is something that wasn't used for the QKD system used during SECOQC. That time only pseudo-random numbers (generated from the computers) were used but especially privacy amplification should use real random numbers.
This device is one of the IQOQI's QRNGs and has been placed at the Alice side of the system.

The QRNG is not used to actively choose the measurement basis as in many QKD systems (as we are choosing the basis passively).
However, for the Software - especially privacy amplification - random numbers are required and instead of just using pseudo random numbers (computer generated random numbers) we decided to use real random numbers by utilizing a QRNG.

Furthermore, for the connection to the Tokyo QKD network we decided to take quite an universal approach, so that our system could be easily modified to work with any other network (further details see chapter 5.)).
In order to achieve this we used the QRNG.

This universal approach was achieved by not actually forwarding the key created by the entanglement source (those keys are only used for necessary secure communication from Alice to Bob) but random numbers.
These random numbers are encrypted at Alice with the keys generated by the entanglement

source by using OTP. These encrypted keys are then sent from Alice to Bob. There they are decrypted with the keys from the entanglement source and then forwarded as keys to the Tokyo QKD network (or it could be any other network).

Therefore, to ensure the safety of QKD we shouldn't rely on pseudo random numbers for this method and so we decided to use the QRNG to get those numbers.

An IQOQI QRNG looks like this from the outside:

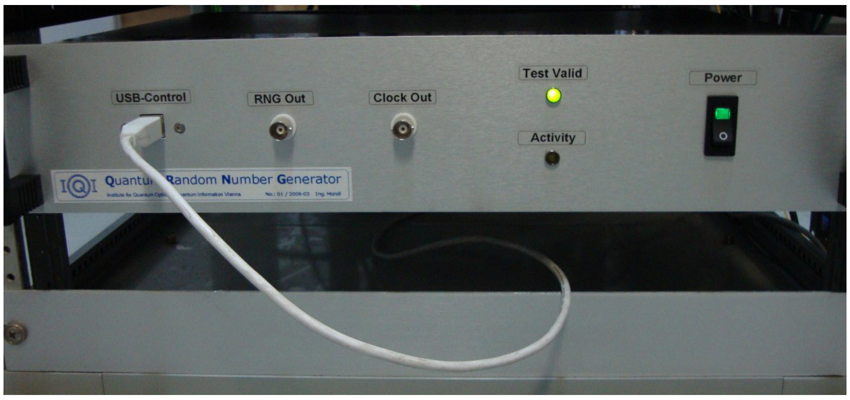

Figure 3.7.: IQOQI QRNG (outside)

This QRNG is simply connected - as can be seen in Figure 3.7 - through an USB cable to the PC at Alice's side. This PC is responsible for QKD stack execution, the storage of the internal keys and forwarding the corresponding keys to Bob's side.
We call this PC Node-Alice. There is an equivalent PC at Bob's side called Node-Bob.

The reason why we decided to use this RNG (random number generator) was mostly that it already existed at IQOQI and is supposed to perform quite well.
Furthermore it uses photon detections to create random numbers which is a good addition as our system uses photons too.

The QRNG works quite simple:
There is a laser source, a 50:50 beam splitter and two APDs which measure the incoming photons.

So the layout looks like this:

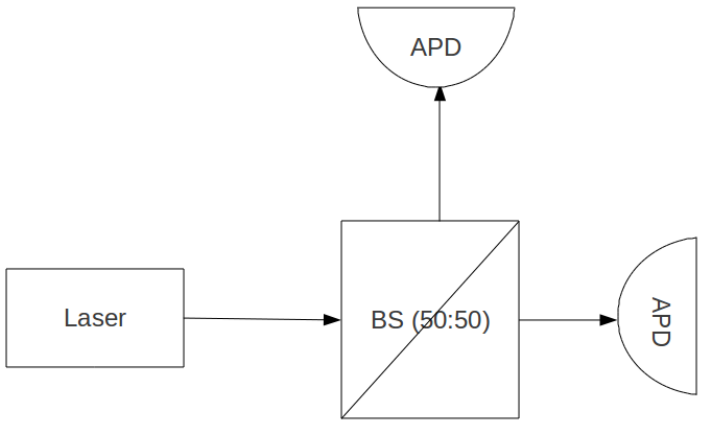

Figure 3.8.: IQOQI QRNG internal layout

The main issue is to ensure that the APDs have the same detection efficiency or, at least, to be as equal as possible.
Therefore, the QRNG also includes a small processor to automatically adjust the BIAS voltages of the APDs to ensure they have the same efficiences.

The main issue with the QRNG was to integrate it with the software. Especially, as our system uses a Linux based operating system (Debian) and the QRNG software only existed for Windows.
However, getting it to work wasn't so complicated as the source code of the Software was available and it turned out the Windows Software uses an open-source library called ”libusb” to communicate with the QRNG. This library also exists for other operating systems like Linux.
So by utilising that library it was quite easy to read random numbers.
A corresponding library called ”libtrng” was created for internal purposes.
The only remaining issue was that the start parameters need to be adjusted manually (as described in an internal document).
Therefore, after turning on the QRNG we needed the GUI to do so but this turned out to be not that much of a problem as the GUI itself was written by using the Microsoft .net Framework and by using the open source equivalent framework called Mono such applications can be compiled and executed under Linux too.
It was only required to change how the program accessed the QRNG (instead of directly calling ”libusb”, it has to use our ”libtrng” library instead).

One small additional program was created that is a service that runs in the background to provide the random number from the QRNG by using TCP/IP and that is how the QKD stack and all other programs get access to the random numbers. Of course, those numbers are only used on the local machine, even though TCP/IP would allow remote distribution but that would be a security issue.

4.). New QKD-Processes “Management”

4.1. Problems with the old Management

During SECOQC the whole startup was done by starting all the necessary processes through shell scripts. There was no real management for the QKD processes which is why I used the quotation marks.

They were started by passing on their configuration parameters through command-line and that was it. This was done through a shell script that was called by a program called ”qdev”. The program should only have been used for authentication of QKD stack messages and messages by the optical management system between Alice and Bob. In the end it also ended up doing the whole management for the optical part of the system. This has been changed now and is explained in chapter 6.) on page 43.

The QKD stack used what is called a pipe. It means that each process passed on its output to the next as input. So qkd_si (sifting) send the sifted key on the the error correction process (e.g. qkd_cascade for cascade). Except for that they only communicated with qdev to do the authentication and exchange messages with the the other QKD device. The old systeem used such pipes for the QKD stack processes.

Here is already one of the main problems with this sort of system: It was quite complicated to exchange information between various QKD-processes except the passed on key. So we weren’t able to easily find out various static values like how many bytes have already been processed or the the error rate or various other values.

Another problem is that for some processes it might be interesting to change various configuration parameters during runtime. Especially for the processes that manage the optical part of the system (see chapter 6.5 for further information).

One other issue was that if you wanted to stop the QKD stack alone, you couldn’t do so because you would also have to stop ”qdev” or actually it would be automatically closed in this scenario. Of course, as ”qdev” was also doing the management for the optical part that would have been stopped too.

Furthermore, in order to adjust some of the processes’ configuration parameters, it was not only necessary stop, change the parameter(s) we want to change and restart the process, but one also needed to pass on all of the unchanged parameters too. However, as most programs were only started through shell scripts those parameters were passed on automatically and it wasn't that problematic but there still are circumstances where it is nice to just stop the process and change one parameter and restart it. For some parameters, of course, it would be even better that we don't even have to stop and restart the process but just change them during the process' execution.

So it was decided that a new central management system for all processes was needed.

4.2. Advantages of the new Management

4.2.1. Centralized Process Management

Because of this centralized process management we are now capable to configure, start and stop processes and also to create, start and stop pipes. through one interface. There is one service (called qkd-ctld) running all the time that takes care of storing and managing all this information (configuration, which processes exists, which pipes exists, ...). So that we can then just issue a command to this service to either change the configuration or, for instance, to start or stop a pipe.

4.2.2. Based on already existing and widely used open-source technologies

In order to store and communicate all this information, it was decided to use an already existing standard interface in open-source applications called DBus (for more information look here [5]). Because of this we are not limited to our own tools alone but we can use already existing tools too. Therefore, it wasn't necessary to e.g. create a tool that displays all the current configuration because there already exist tools that can display this information (and more). We only needed one (which is called qkd-ctl) to easily store the configuration, edit it and issue start and stop command. Those existing open-source tools can also be used for testing purposes. They were extremely useful in some tests regarding the new optical management (see chapter 6.))

4.2.3. Better communication between processes

This was one of the main reasons for the new management. With the old system the possibility to exchange information between processes was extremely limited. We either had to use the standard input and output or use something similar to them called "named pipe"[1] under Linux based operating systems. One of the first problems is that for different information you need individual files. Furthermore, you need to know the length of the information which means that you need some kind of header that has always the same length and stores the real length of the data. The biggest issue, however, is that you can only write from one process and read from another. You cannot send information backwards. For the new optical management (for details read chapter 6.)) to get rid of this limitation was really helpful. All of this is simply accomplished by using processes that take advantage of the new management. Of course, named pipes are still

[1]It is a FIFO, which means whatever is written to it first is read first from it, based "file"

used because for some data (e.g. key data) they work fine (and had already been implemented anyway) but for small information data the new management is used.

4.2.4. Required for Tokyo QKD network

This is related to the advantage of better communication between process, see chapter 4.2.3. For the Tokyo QKD network it was necessary that we also inform their network about some statistical values from our system to be used for statistics during the Tokyo QKD network demonstration. This requires, e.g. the current error rate and the bits after the sifting process. This information was spread over various processes and needed to be accumulated and forwarded to the Tokyo QKD system. Since we now have the new management capable of handling this type of data exchange, it was quite simple to do so.

4.3. How the processes communicate

This is just to get a feeling of how the processes communicate using the new management system and which ones are used in a normal startup (excluding the processes used for the optical management, those are explained in chapter 6.) on page 43).

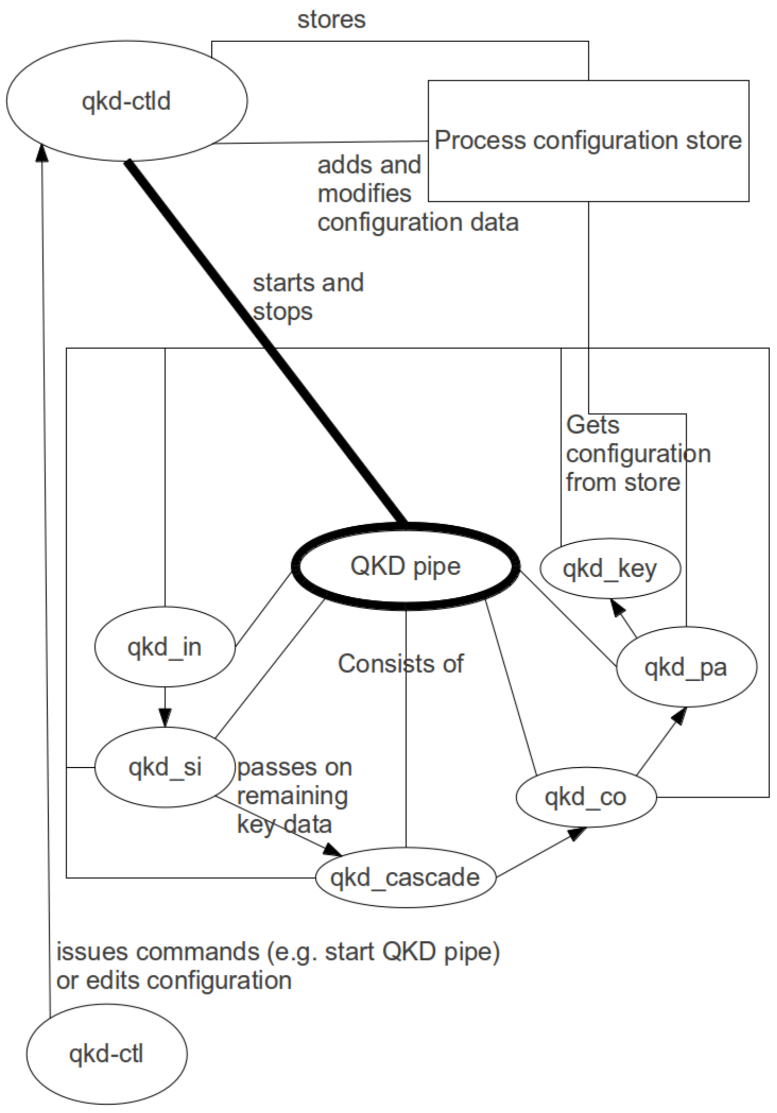
stores
qkd-ctld
Process configuration store
adds and modifies configuration data
starts and stops
Gets configuration from store
QKD pipe
qkd_key
qkd_in
Consists of
qkd_pa
qkd_si
passes on remaining key data
qkd_co
qkd_cascade
issues commands (e.g. start QKD pipe) or edits configuration
qkd-ctl

Figure 4.1.: New management - Process diagramm

This diagramm includes all the processes that were used for the QKD stack in Tokyo except "qdev-q3pdev-bridge" and "qkd_queue".
Every process of the "QKD Pipe" has a named pipe communication with "qdev-q3pdev-bridge" which is what remains from the old "qdev" (described in section 4.1).
However, "qdev-q3pdev-bridge" now only does authentication between Alice and Bob (and handles log messages from all QKD stack related processes).
"qkd_queue" will be explained in section 4.5 but it would basically go between "qkd_co" and "qkd_pa".
As for the processes in the diagramm, here a small explanation about what they do:

qkd-ctld	Used to store the process configuration data and also starts and stop processes and modifies the configuration data
qkd-ctl	A tool that is used to issue commands to qkd_ctld. Those commands are used to start and stop processes and to modify configuration data.
qkd_in	receives the raw data from the QKD device. So it gets the information about which detector clicked.
qkd_si	does the sifting of the raw data
qkd_cascade	uses cascade for error correction of the key data
qkd_co	verifies that the output of the error correction is correct and informs the optic management about the current QBER.
qkd_pa	used for privacy amplification
qkd_key	forwards the final key to the key-store

4.4. How to use the new management system

To use the new management system qkd-ctld has to run and then we can control everything by using qkd-ctl. We have to make sure that both processes (and also the qkd-processes themselves) are connecting through the same DBus session which isn't an issue as long as we use the same shell script or terminal. Otherwise we would have to set the DBus-session address before starting the qkd-ctld and before using any other processes by using something like this:
export DBUS_SESSION_BUS_ADDRESS=tcp:host=localhost,port=9875

Before starting qkd-ctld we have to make sure that a DBus (to which we can connect) is running. This can be achieved by executing

eval ‘dbus-launch --config-file=/path-to/configuration-file.conf --auto-syntax‘
but first you need a configuration file (here it would be /path-to/configuration-file.conf) for the DBus (for more information regarding this look at [5]).

Afterwards, you can use qkd-ctl to create a process, edit its configuration, create a pipe, add processes to the pipe and start the pipe like this:
Create a process: qkd-ctl process-create PROC_NAME /path-to/process

Set configuration property:
qkd-ctl process PROC_NAME prop PROPERTY_NAME=PROPERTY_VALUE
It is possible to set more than one property at once by seperating them with space

Create a pipe: qkd-ctl pipe-create PIPE_NAME

To remove all processes from the pipe (in this case to make sure that it is empty):
qkd-ctl pipe PIPE_NAME clear

Add processes to the pipe: qkd-ctl pipe PIPE_NAME append PROC_NAME

Start the pipe: qkd-ctl pipe PIPE_NAME start

For older processes that don't use the new configuration storage it is required to configure them by using command line parameters which can be done like this:
qkd-ctl PROC_NAME param-append COMMANDLINE_PARAMETERS

If we want to stop the pipe we just need to use: qkd-ctl pipe PIPE_NAME stop

In order to find out the command-line parameters of a process and what they do, we can just type the process name into the terminal and add " --help" (or in most cases just the name should be enough).

In order to find out the new property names for the configuration through the configuration store, you can do the same but have to use the "long" paramter (the one with the "--") and ignore the "--".
It might also help to take a look at configuration samples provided with the source-code or to take a look at the qdev_managed.sh on the QKD nodes.
There should also be a documentation regarding the configuration paramters at the AIT qkd stack Wiki (https://sqt.ait.ac.at/software/wiki/qkd-stack/ [6]) in the future.

Two examples for parameters:
-) qkd_cascade - "type": defines if I use parallel mode cascade or not
-) most processes - "unix-socket": used to pass the key material on to "qdev-q3pdev-bridge" for

authentication

4.5. Invention of the so called qkd_queue modul

One more issue had to be taken care of. The issue is that for slow keyrates the optical management still needs to be capable to respond to changes in the QBER fast.
Otherwise it could happen that we have 0 keyrate or nearly 0 keyrate for quite some time.
So we needed to use ”smaller” block sizes after sifting to achieve new QBER values from qkd_co fast enough.

But on the other hand the qkd_pa (privacy amplifcation) needs a certain block size (bits to process) to be secure.
This is a general issue but even more so now as the crystals don’t work very well anymore (more regarding this in chapter 7.)).

So the following approach was used:
After sifting (qkd_si) we use small block sizes and allow ”fast” changes for the QBER value but after qkd_co (confirmation of correct done error correction) we ”merge” blocks that passed the error correction
until we have enough bits to pass it on to the privacy amplification.

This actually has a few more advantages:
1.) Even with good keyrate during SECOQC we were at the limit to react ”fast” on changes. Since the new optical management is now handling more errors (see chapter 6.)), it needs to react faster than previously.

2.) It can happen that the QBER suddenly increases to high values (but still lower than the one for a zero keyrate). This might result into the remaining bits after error correction to be below the required secure bits for privacy amplification. As we are now waiting to get enough bits, this is not an issue anymore. The system will just wait until it has enough bits again.

So this module is quite helpful and has to be placed between qkd_co and qkd_pa. It uses the configuration parameter ”key-length” to configure the required bits before passing the key on to qkd_pa. The value we used during the UQCC 2010 demonstration was ”300.000” bits.

4.6. How to simply do a normal startup

In order to be able to start and stop everything easily without the need to know much about the rest of the system and without entering to many commands, there exist two scripts on Node Alice that will start and stop everything required to get the system to work and forward the key data to the Tokyo QKD network (explanation on how the key is forwarded is done in chapter 5.)). Those two scripts are nodectl and linkctl:

4.6.1. nodectl

nodectl has 3 options:

start	Starts the local nodes where the key data is stored until it is forwarded to the Tokyo QKD network. It also starts the DBus required for the QKD stack processes and at last the process that takes the keys from the local store and forwards them to the Tokyo QKD network (further information in chapter 5.)).
stop	stops everything that was started with the option ”start”
status	displays which processes related to nodectl (see ”start” option) are currently running

4.6.2. linkctl

linkctl has 6 options:

start	Starts the QKD management, creates all the pipes and configuration for the QKD stack modules, the optical managment modules and ”qdev-q3pdev-bridge” for a full (normal) startup. This means the optical management does all the required first startup routines and it also does the normal routine and react to QBER changes.
quick	it will directly start QKD key generation (so no startup routines) but it will do all the normal routines and react to QBER changes. This mode should only be used after running ”start” at least once before since at least one of the QKD devices was powered off.
counter	This mode’s main purpose is to do optical alignments before we can start with QKD key generation. Therefore, it doesn’t do any startup routines, nor does it do the normal routine or react to QBER changes. It mainly starts everything so that the detector counts are displayed in the monitoring application (for more information see [7], [8], [9]). this mode is helpful for entanglement source alighment.
mode	This is only for eventually future optical routines. Currently it isn’t required because of ”quick” and ”counter” but with this it is possible to pass on a number which specifies how the optical management shall react (e.g. if it shall react to QBER changes or not; for more more details see chapter 6.)).
stop	stops everything that was started by using the option ”start”, ”counter”, ”quick” or ”mode”
status	displays which processes related to linkctl (see ”start” option) are currently running

5.). Integration into Toyko QKD network (UQCC 2010)

5.1. Tokyo QKD network

First I want to explain a bit the Tokyo QKD network (additional details about the network can be found in [1]).
It was quite similar to the SECOQC network demonstration in Vienna but, of course, with newer technology.
There were various teams included and they were connected like this:
.

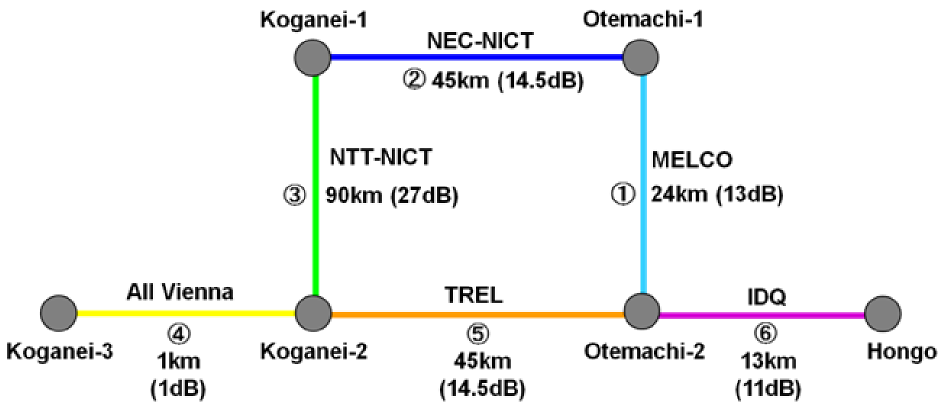

Figure 5.1.: Logical link configuration of the Tokyo QKD Network., from "Field test of quantum key distribution in the Tokyo QKD Network" [1, Page 10391 Fig. 1. (b)]

Our system was called the "All Vienna" and as the picture shows we actually had quite a small distance to overcome for QKD. But regarding this more in chapter 7.).
What might be to add is that our link was actually a loop cable that was going around outside of the building and coming back to same room.
Originally it should have been two different buildings but they needed that room for other experiments.

The whole network is divided into three layers:

1. Quantum layer:
   This layer is where the keys are securely distributed through QKD. So this is the layer where the QKD devices are operating.
   These devices push their keys to the "KeyAgents" (also called "KM Agents" in 5.2) which are part of the Key Management Layer.

2. Key Management Layer:
   This layer is where the so called "KeyAgents" are used.
   These agents store the key material at the individual nodes and provide it for communication to the "communication layer".

3. Communication Layer:
   This layer does the actual communication by encrypting it with the keys stored in the KeyAgents.
   It includes one important server and that is the so called "Key Management Server". This one is responsible to determine which path to use for data communication.
   For instance, in the Tokyo QKD network if the TREL link fails (e.g. because of an eavesdropper) and we need to send data from Koganei-2 to Otemachi-2, the key management server determines (depending on how much key data the KeyAgents have stored) to send the data over NTT-NICT, NEC-NICT and MELCO, instead of the TREL link.

Those layers are further illustrated in this picture: .

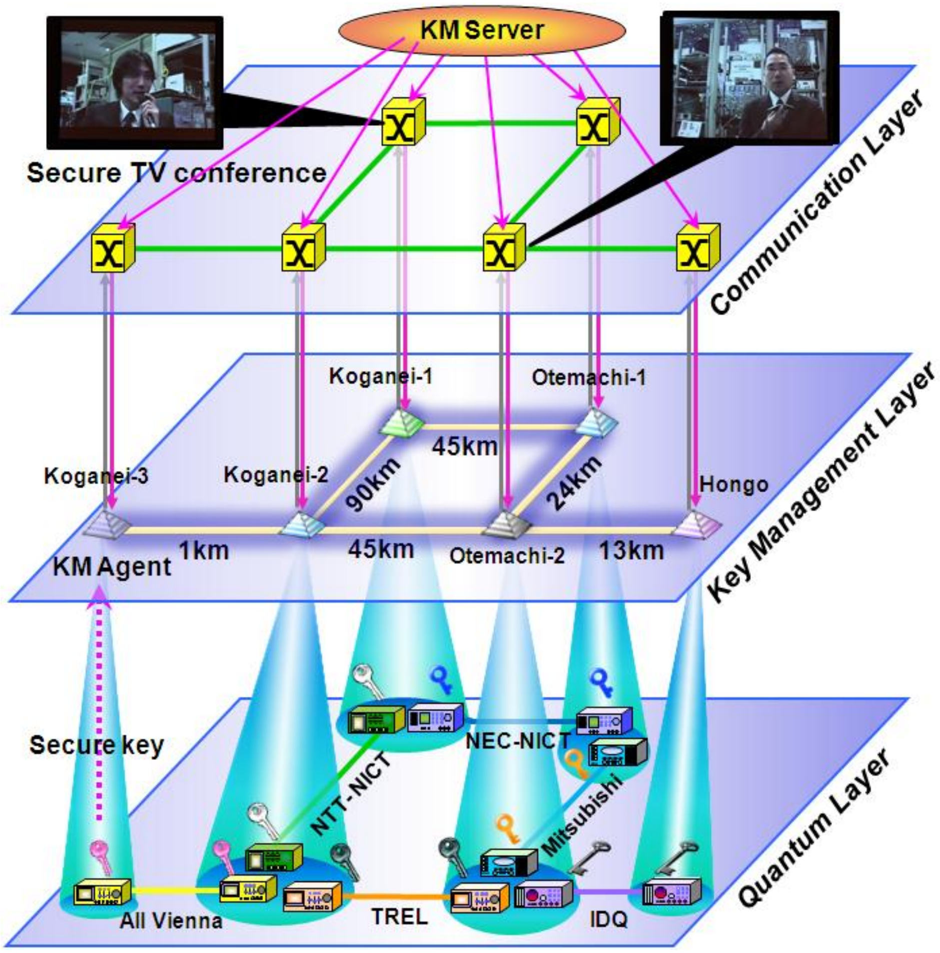

Figure 5.2.: "Three-layer architecture of the Tokyo QKD Network", from "Field test of quantum key distribution in the Tokyo QKD Network" [1, Page 10393 Fig. 2.]

One last picture that shows the wiring of the systems like network connections, optical fibres (for QKD or also for management purposes):

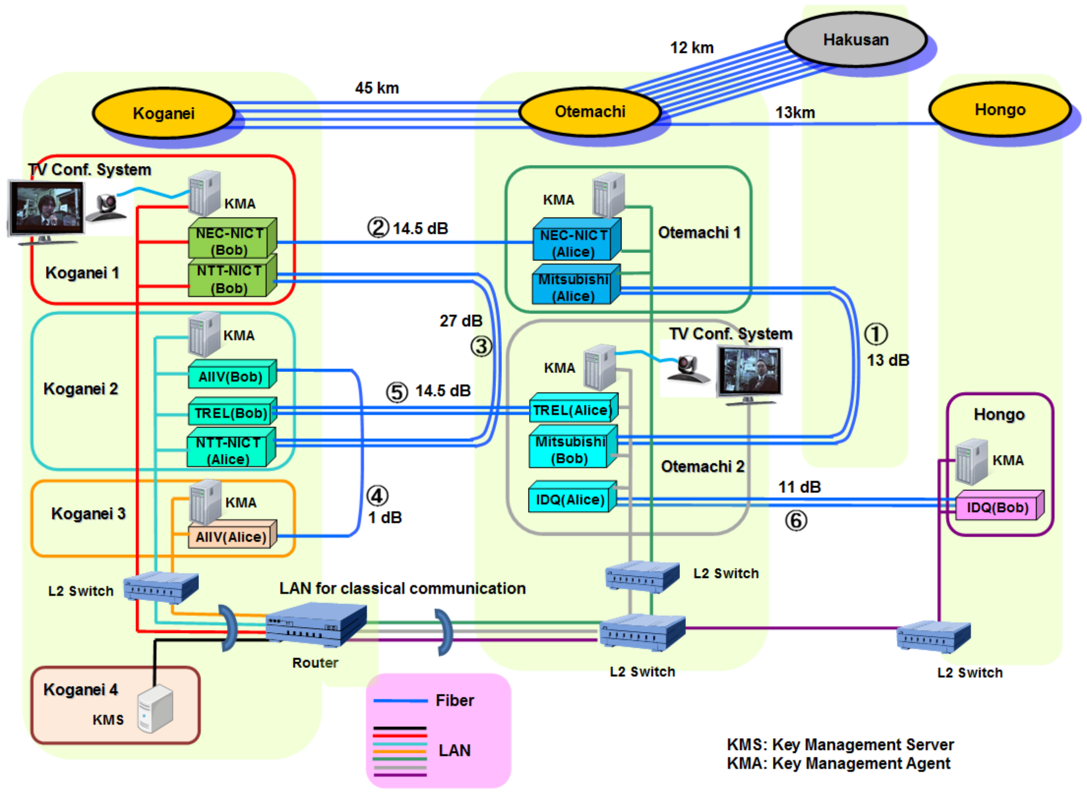

Figure 5.3.: "Wiring diagram of the Tokyo QKD Network.", from "Field test of quantum key distribution in the Tokyo QKD Network" [1, Page 10393 Fig. 3.]

5.2. Connection with our system

Out system is providing the Tokyo QKD network with keys by using modified SECOQC node software and an additional program called "q3p-mqr".
What we do is the following: We use our QKD devices to distribute/produce keys and store them in the local node software.

This local node software is a modified version of the SECOQC nodes as it includes DBus for configuration and they export statistical data in the same way as the new "New QKD-Processes “Management” (look at chapter 4.)).
However, the main difference is that they are now used in the so called "direct mode" which only allows direct connections from Alice to Bob (for communication) and doesn't include routing over other SECOQC nodes.
This is more than sufficient for what we need them at the Tokyo QKD network and it reduces latency.

The stored keys in those nodes are then used for needed secure communication between Alice and Bob - e.g. for various exchange messages from the optical management software (look at 6.)).

Those nodes, however, now have one more task and that is related to the "q3p-mqr" program.
The program "q3p-mqr" reads keys, accumulates statistical data and forwards both to the KeyAgents.
And exactly that is where the local node software has an additional feature and that is to provide keys for the "q3p-mqr" process.

The local node software at Alice reads data from the QRNG (see 3.2.5) provides it for the "q3p-mqr" as keys at Alice, encrypts it through the stored available keys using OTP, sends the encrypted data to Bob, where the data is decrypted using the stored available keys and Bob provides that decrypted data as keys for "q3p-mqr".

This is only done after the stored key data at the local node have reached a certain limit (can be configured using command line parameters). Then all the data, except a certain minimum amount of local stored key data (also configurable) is reached.
Afterwards, the local nodes are filled with key data again until the limit is reached.

The disadvantage of that system is that the Tokyo QKD network only gets data after a certain amount of time but, of course, it is quite a lot then.

However, there is one nice advantage of this method as it is usable for any other QKD network software the only thing that would need to be changed is "q3p-mqr" as this has the only

connection to the Tokyo QKD network and even here we would only need to change how the data is send there, not how we read it from our local node software.
Which means that for our internal communication we don't have to rely on the "external" QKD network anymore.

This should make future adjustments for new QKD networks quite easy.

So the connection to the Tokyo QKD network is like this:

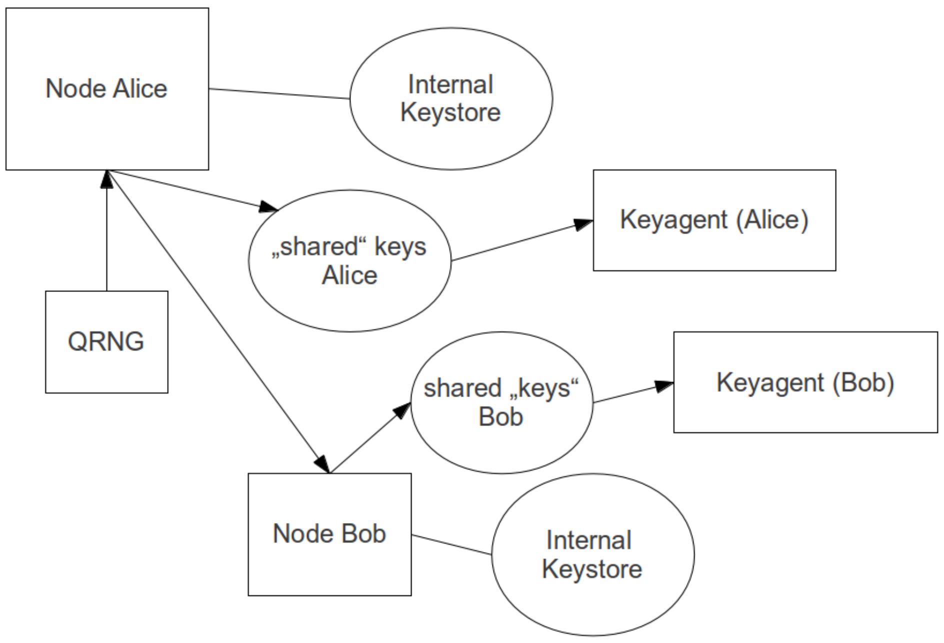

Figure 5.4.: Connection scheme of our system to the Tokyo QKD network

6.). Better Optical Management

6.1. Uses features from the new QKD-Processes management

For information regarding the new QKD-Processes management look at chapter 4.) The new optical management now uses the new QKD-Processes management to start the required processes, set certain configuration parameters and to export and read data between processes. This also allows for a way better structure and also better error handling.

The optical management system now consists of 4 processes on the nodes and the so called PolCtrl-Software which exists in different forms for Bob and Alice and are stored on FPGA boards.

The PolCtrl-Software is immediately running once the system is activated.
Except that it might be necessary to load the current software (if it isn't already flashed into permanent memory) into the RAM by using Xilinx utilties and an USB connection to a PC.

The four processes are "optic_master" and "optic_ctrl" at Node Alice and "optic_slave" and "optic_tunnel' at Node Bob. Those are manually started by using the new QKD-Processes management.
In the old optical management, all of this was together in one process that normally should not have any connection to the optical management.
However, because of time issues during SECOQC and because it wasn't so easy then to export data to a different process (there was no startup-management and therefore no DBus data exchange) all of the optical management was put into the "qdev" process as that one had access to the QBER value which was required for the error management.

They are started through the "linkctl" shell script (which is further explained at chapter 4.6).
Each of those programs has various configuration parameters which will be explained in chapter 6.5.
Some of those parameters can also be adjusted during program execution and for which parameters this is valid will also be mentioned at chapter chapter 6.5.

For now this diagram shows how the individual processes are connected and what data is exported and read from which process. This is necessary to explain how the error handling works (explained at chapter 6.4) which is quite important as this is one of the differences between the new and the old optical management system and also to better understand what the individual processes do.

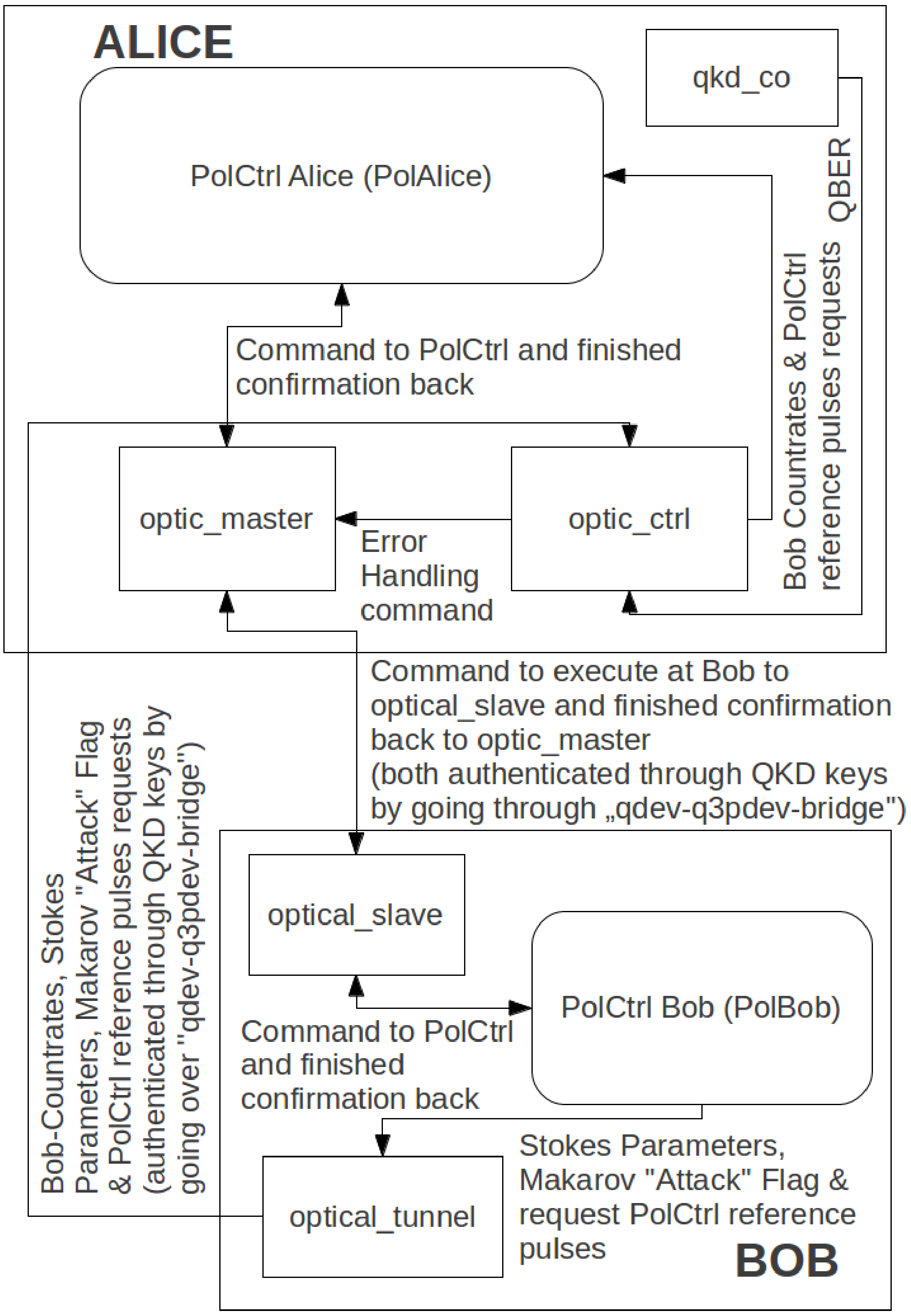
ALICE
qkd_co
PolCtrl Alice (PolAlice)
QBER
Bob Countrates & PolCtrl reference pulses requests
Command to PolCtrl and finished confirmation back
optic_master
optic_ctrl
Error Handling command
Command to execute at Bob to optical_slave and finished confirmation back to optic_master (both authenticated through QKD keys by going through „qdev-q3pdev-bridge")
Bob-Countrates, Stokes Parameters, Makarov "Attack" Flag & PolCtrl reference pulses requests (authenticated through QKD keys by going over "qdev-q3pdev-bridge")
optical_slave
PolCtrl Bob (PolBob)
Command to PolCtrl and finished confirmation back
optical_tunnel
Stokes Parameters, Makarov "Attack" Flag & request PolCtrl reference pulses
BOB

Figure 6.1.: Optical Management Processes and interaction

The "optic_ctrl" process also forwards various measurement/statistical data, that it accumulated, to a monitoring application through TCP/IP. This monitoring application can also store that data in a database.
Of course, it would also be possible to write a program/shell script that stores that data directly into the database without it first being processed by the monitoring program but in most cases someone wants to be able to monitor the data directly too and therefore so far we never did store the values into the database without the monitoring application.

Explanation of the individual processes:

1. **optic_master:**
   This is the most important process of the optical management.
   It is responsible for coordinating everything. This program sends commands to the individual parts of the systems. Therefore it lets each individual part know what to do at the current time.

   When you start the system with "linkctl start" it first does a full start up sequence and only afterwards it enters normal operation (more information at chapter 6.3).
   Normal operation means that it repeats various optical adjustments all the time. However, this can be interrupted by an "Error Handling Command" from optic_ctrl.
   It then starts the correct error handling procedure as fast a possible (more information at chapter 6.4).

2. **optic_ctrl:**
   The main function of this process is to monitor the current QBER value (from qkd_co, not qkd_queue, because it has a faster refresh rate) Futhermore, this process uses the DBus features of the new management system to keep an eye on the QBER, details about this at chapter 4.2.3 and decides if there is an issue/error with the system and what type of error.
   This information is then transmitted by using the DBus interface (see chapter 4.)) to optic_master which then initiates possible counter-measures (more information at chapter 6.4).

   Furthermore, it is responsible to receive various data (authenticated through QKD keys) from Bob (through optic_tunnel).
   This data contains:

   a) Statistical data:
      This data is just used to get current information about the system.
      It contains the countrates of Bob's individual detectors and the Stokes parameters obtained by PolCtrl procedures.

b) "Makarov Attack" Flag:
This one is used to inform the optic_master that a "Makarov Attack" occured and we need to respond accordingly (further details at chapter 6.4).

c) PolCtrl reference pulses requests:
The PolCtrl procedure at Bob requires reference pulses from Alice and therefore we have to request them from Alice.
This is done over authenticated communication (through QKD keys) and therefore passes through here and is then forwarded to PolAlice (more information at chapter 6.2).

3. **optic_tunnel:**
This process just forwards data from Bob to Alice (there it is received at optic_ctrl) over authenticated communication (through QKD keys).
This data was already described for optic_ctrl.

4. **optic_slave:**
This process only receives commands from Alice (over authenticated communication through QKD keys) and then forwards them to PolBob which then starts the corresponding procedure.
Once the procedure at PolBob is finished, it informs optic_slave which then forwards this message to Alice's optic_master.

6.2. Procedures that do actual adjustments

Some of the procedures described in this chapter have been modified since SECOQC and one has been added, those differences are also explained here and also why they were modified.

Most of the procedures are executed through the FPGA boards PolAlice and PolBob. Those that are executed there involve optical elements of the system.
Those procedures are invoked by sending commands over UDP (e.g. by using netcat) using port 10030/11030 from Alice/Bob to PolAlice/PolBob (further information regarding ports and how to invoke procedures manually can be found in internal documents written by Alexander Treiber [9] or the manual [8]). However, the two procedures find_delay and find_window are not executed at PolBob or PolAlice but directly at Bob (responsible for reading the detector clicks and producing the key material of those).
In order to fully understand the system it is important to know what procedures exist and what they do.

1. **SourceStab (Source Stabilization):**
   This one is used to keep the entanglement source stable during operation. This is necessary because the source is free-space and therefore prone to changes due to e.g. heat.
   It basically tries to maximize the countrate of each individual Alice detector.
   It uses the hill-climber algorithm to do so which means it measures a value, changes the positioning of the optical element (this is called a step) by a certain value (this value is called step width), re-measures and if it got better, it goes further in that direction, otherwise it tries in the other direction.
   If that direction also results in a worse value, then the one in the middle (in this case the starting point) was the best setting.

   There are three elements that are controlled by SourceStab and each one has two axis for adjustment which means that there are six degrees of freedom which are controlled through the channels of the piezo mounts AM-M100 from Newport with two tilt axes.

   The hill-climber algorithm uses an average of multiple points to determine the next step. It uses 10 points for the channels that get the data from Alice and 20 that get the points from Bob int NORMAL mode. Both sides use 10 points in QUICK mode.

   The available channels to control the source stabilization are:

Channel	**Piezo-Component**	**Data source**
1 & 7	Laser mirror, axis 1	Alice
2 & 8	Laser mirror, axis 2	Alice
3	810nm coupler, axis 1	Alice
4	810nm coupler, axis 2	Alice
5	1550nm coupler, axis 1	Bob
6	1550nm coupler, axis 2	Bob

   The laser mirror axis have two channels because they previously (SECOQC) they were called at the beginning of the procedure but now they are called at the end of the procedure (or not at all, see commands below). This procedure can be invoked by sending either of the following commands to PolAlice.
   There are more than one command to better control what exactly SoureStab does:

   a) START:
      This command does a normal SourceStab with all channels and small step width.
      The step widths actually depend on what we are adjusting (currently set to 2 for 810nm coupler, 2 for 1550nmn coupler and 1 for laser mirror).
      This mode uses 10 numbers to average for all elements that use Alice's counts and 20 numbers for those that use Bob's counts.

b) QUICK ALL:
It has a bigger step width and therefore doesn't take so long (the current step widths are 6 for 810nm coupler, 8 for 1550nmn coupler and 4 for laser mirror).
This mode uses 10 numbers to average over the counts on both sides (Alice's counts and Bob's counts).

c) QUICK ALICE:
This option was added after SECOQC. It uses the quick mode settings (like QUICK ALL) for averages and step sizes.
This one was added as it is useful for a first start up where the source might be quite misaligned (necessary to ensure that enough counts are available for find_delay and find_window; see next page on information for those procedures). It does the source stabilization only for the 810nm coupler (channels 3 and 4).

d) QUICK BOB:
It was also added after SECOQC and for the same reason as "QUICK ALICE".
This only does channels 5 and 6 (1550nm coupler) which require the Bob counts. Furthermore, it uses the quick mode settings (like QUICK ALL) for averages and step sizes.

e) QUICK LASER:
This uses quick mode settings (like QUICK ALL) for averages and step widths and only does channels 1 & 2 (equivalent to 7 & 8) for the laser mirror.

f) QUICK BOTH:
This uses quick mode settings (like QUICK ALL) for averages and step widths and does channels 3 & 4 (810nm coupler) and 4 & 5 (1550nm coupler).

g) QUICK ALL:
This uses quick mode settings (like QUICK ALL) for averages and step widths and does channels 3 & 4 (810nm coupler), 4 & 5 (1550nm coupler) and 7 & 8 (laser mirror).

h) STOP:
Stops SourceStab during execution. This is required for a better error handling, see chapter 6.4.

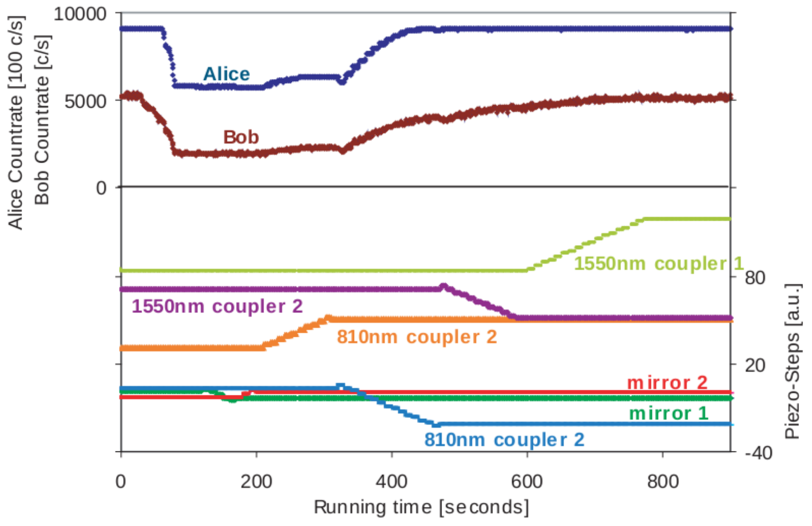

Figure 6.2.: "SourceStab under test: at the beginning all six channels have been changed manually to decrease the count rates. On the lower half, one can see the steps of the piezo-mounts driven by PolAlice to retrieve the original count rates.", cited from [7, Figure 6.2 Page 33]

2. **Crystal temperature:**
   This procedure just measures the crystal temperature and returns it (in the normal operation it is returned to the optic_master process).

   This procedure is invoked by sending "TEMP" to PolAlice.

3. **find_delay:**
   This procedure is actually a program at Bob - in contrast to the others that are procedures on the PolCtrl FPGA boards.
   The delay between the trigger pulse (1610nm) that is used to open the detector gates and the pulse that contains the key information (1550nm) depends on the fibre length between Alice and Bob.
   Therefore on start up we have to figure out the correct delay.
   This program is used to automate that process.

   Furthermore, it is used to make sure that during normal operation we also keep track of small delay changes due to temperature changes.
   There were no changes in this procedure since SECOQC and it is explained in detail in [7, Chapter 6.4 Page 48].

This program is called by using either ”-g” or ”-l” by the optic management.
”-g” is used during the start up which checks the whole delay range (10ns) for the correct delay.
”-l -w” is repeatedly used during the normal QKD operation to search around the current delay position for a better delay because of possible delay changes due to temperature. This also adjusts the the auxiliary window (for more information regarding this see item 4 of this enumeration) correctly to the new delay because of the use of the parameter ”-w”. The full parameter list can be shown be using the command line parameter ”-h”.

4. **find_window**:
   This procedure is like find_delay a program at Bob.
   The origin of this module is that during the tests for SECOQC in the laboratory where a detector build by the KTH Stockholm with a fixed and broad gate width (about 5ns) was used (see also [7, Chapter 6.4.2 Page 51]).
   Therefore, an additional auxiliary window was used by using a time-tagging-unit (TTU).

   We can only use detection events during a certain time-window.
   This time-window isn't fix, of course, and therefore needs to be found by using find_window. Currently it is set to use a 2.5ns window for all detectors - though it could be way smaller. It is actually capable of finding the best width by itself but in the last tests we noticed that over longer periods the counts went down which was probably because in the updating of the optical management we forgot to re-adjust the window when adjusting the delay values (this should be fixed by now as we added the ”-w” parameter to find_delay).
   However, we were not able to verify this result before the UQCC 2010 demonstration and therefore we decided to use a fixed higher window size for the Tokyo QKD demonstration at the UQCC 2010.

   What is really interesting about this software window though is that it can also be used as counter-measure for some of the newer ”Makarov Attacks”. For further information see chapter 8.).

5. **PolCtrlInit (PolCtrl Initialization):**
   It initializes the polarisation controllers by applying voltages at the middle of the range (those controllers are required for StateAlign, see item 7 of this enumeration, and, of course, also by PolCtrl. It sets the default values required for PolCtrl procedures and does the first normalization of the receiver diodes. The normalisation is required before we are capable of calculating the stokes parameters, for the reason check out [14].

   For further information regarding this (and PolCtrl in general) take a look at [7, Chapter 6.3.3 Page 43]).

   PolInit is invoked by sending ”START PolCtrlInit” to PolBob.

6. **PolCtrlNorm (PolCtrl Renomralization):**

As already mentioned in PolCtrlInit before calculating the stokes parameters we need to renormalize the receive diodes.
This is what this function does and it is explained in detail at [7, Chapter 6.3.3 Page 43].

PolCtrlNorm is invoked by sending ”START PolCtrlNorm” to PolBob.

7. **StateAlign (State Alignment):**
Because the birefringence in the fibre causes polarization changes, we have to compensate them by using polarisation controllers.
We have two polarization controllers: one for the P (+45°)/M (-45°) basis and one for the H/V basis (look at Figure 3.1 for details about the layout). Even though our entanglement source itself generates

$$|\phi\rangle = \frac{1}{\sqrt{2}} \Big(|H_{810}\rangle \otimes |H_{1550}\rangle + e^{i\phi} |V_{810}\rangle \otimes |V_{1550}\rangle \Big)$$

We are actually adjusting it to be a

$$\left|\psi^-\right\rangle = \frac{1}{\sqrt{2}} \left(|H_{810}\rangle \otimes |V_{1550}\rangle - |V_{810}\rangle \otimes |H_{1550}\rangle \right)$$

before coupling into the quantum channel.

StateAlign is a procedure that tries to align this state.
StateAlign only works correctly if the alignment is already correct at Alice (meaning the photons are already aligned correctly before they are sent to Bob over the quantum channel). This needs to be done manually, there is no automatic adjustment for this as it isn't necessary to adjust this for long-term stable QKD and only at the beginning.

We are basically using the hill-climber algorithm to get a correct state. However, in contrast to source stabilization we are looking for minima and not maxima.

So what we are doing is the following:

a) Only trigger photons that are measured with a P (+45°) polarisation at Alice.
b) Use polarisation controller to achieve a minimum at the P detector at Bob.
c) Only trigger photons that are measured with a V polarisation at Bob
d) Use polarisation controller to achieve a minimum at the V detector at Bob.

Now if the source was already correctly aligned at Alice, we should now have a perfectly aligned state.

The reason why P comes before V is because of the positions of the polarization controllers. By changing the polarisation controller for P/M basis, we would also also introduce changes to the V/H basis and therefore we need to first align the P/M basis.

The main problem with this procedure is that it takes quite some time and we cannot produce any keys during that time which is why we are only doing it at the start up of the system or when it is absolutely necessary (the optic management system decides if it is necessary, see chapter 6.4).
During normal operation the PolCtrl (polarisation control) is taking care of keeping the correct state.

Details about the originals StateAlign procedure can be found in [7, Chapter 6.2 Page 34ff].
Originally, this procedure was run twice because one time wasn't enough to produce a promising result but the procedure was improved now and therefore one time is enough.

The main problem with the original procedure was that it stopped alignment after a certain threshold was reached which was accepted as minimum value. This value was adjustable but it is kinda pointless to have a system that aims for automatic adjustments to use a fix limit when to stop.
The improved procedure now aligns until we find a minimum (even if it might take a bit longer but the alignment is correct afterwards).

Furthermore, the original StateAlign used two different step sizes. It used bigger ones at the beginning and smaller ones after the detection counts fall below a certain value. The old procedure interpreted this as getting closer to the minimum.

This concept to use bigger steps at the beginning is good to decrease the required time of the procedure. However, we again have a limit when it starts to use smaller steps. Therefore, the time this procedure requires is longer than necessary because it is likely to be quite a while until we reach the minimum then with the smaller steps.
So we decided to use a different approach which is to use the big steps until we find a minimum and then switch to the smaller steps and try again to find a minimum.
By doing it this way, we are way closer to the real minimum when we switch to the smaller steps.

Furthermore we increased the step sizes for both (small and big steps) as the step sizes for the small steps were so small that there was more fluctuations in measurement than real differences. The step size of the big steps was increased to decrease the time we need to find the rough minimum. This doesn't decrease the quality of the alignment because afterwards the procedure tries to find the real minimum by using the small step size.
However, we can't just use an extremely big step size, as we would then have to switch to

the small step size too early and the performance gain would be lost again.

With the optimized step step sizes and the changes on how the procedure finds the minimum by using those different step sizes, we were able to decrease the time this procedure requires quite a lot.
However, we also decided that it was better to increase the number of values to calculate the average for current position (to decrease the influence of the noise and therefore to decrease the need of repeating the StateAlign procedure).

The StateAlign procedure is invoked by sending "START PolAlign 2" (for P/M basis) or "START PolAlign 0" (for H/V basis) as command to PolBob.

Furthermore, after both bases are aligned (meaning after "START PolAlign 2" and "START PolAlign 0") this procedure will automatically measure (and store) the target state for the polarisation control procedures.

8. **PolCtrlCycle (Polarisation control cycle):**
   This procedure requests the reference pulses (H and P) from Alice, measures their current state, calculates how to get to target state and does the corresponding changes in the polarization controllers.
   One of the changes since SECOQC is that the algorithm to get there has been improved.
   Furthermore, if the procedure didn't get close enough to the target state with one try, it requests again the reference pulses and tries again.
   If it isn't near enough again, it tries once more until at most 9 repeats (a total of 10 tries).

   For details on the exact routine of PolCtrl read [7, Chapter 6.3 Page 40ff].

   PolCtrlCycle is invoked by sending "START PolCtrlCycle" to PolBob.

9. **"Makarov Attack" Check:**
This is the procedure that was added after SECOQC. It was added because of discoveries made by Vadim Makarov (for details see chapter 8.)) and for the system to have the possibility to counteract the attacks described in chapter 8.). This procedure isn't invoked manually but always executed in the background, once PolCtrlInit was executed.
It checks the voltages of the polarisation diodes as often as possible. It takes those voltages and compares it with a voltage limit which can be set per diode. Per default this limit is at -1.0 V for all diodes.

With this it is possible to counteract against the blinding "Makarov attack" by using high intensity cw-light (further details see chapter 8.1). It should also be possible to detect the thermal blinding attack this way (see chapter 8.2).
Because of the system layout (see Figure 3.1) 5 % of the incoming light always goes to the polarisation control and therefore we can use this to measure if the intensity of the quantum channel is to high which would mostly likely be because of a "Makarov attack".

As soon as one diode reaches the limit we sent a "MAKAROV ATTACK" signal/flag (exact message is: "MAKAROV-OCCURRENCE TRUE") to optic_tunnel which forwards it to optic_ctrl where it is then handled as an error and informs optic_master to take action by using the corresponding error handling command (further information can be found in chapter 6.4).

There is also a signal that informs optic_ctrl through the same way that the attack has ended (same message but "TRUE" is replaced with "FALSE"). This is done once ALL diodes are below the limits again. Those limits can be set individually for each diod. The individual voltages per diode have the advantage that they could be manually tuned to each diode, so that the limit is near the normal operating value with only little leeway.

These values can be adjusted by using

Command	**Description**
SET MAKAROVLIMITS	Sets the limit for all diodes
SET MAKAROVLIMIT0	Sets the limit diode number 0
SET MAKAROVLIMIT1	Sets the limit diode number 1
SET MAKAROVLIMIT2	Sets the limit diode number 2
SET MAKAROVLIMIT3	Sets the limit diode number 3
SET MAKAROVLIMIT4	Sets the limit diode number 4
SET MAKAROVLIMIT5	Sets the limit diode number 5

6.3. Optical management sequences

After explaining the procedures, it is now necessary to explain in which order they are invoked during start up and during the normal operation (as long as no error occurs).

The start-up sequence looks like this:

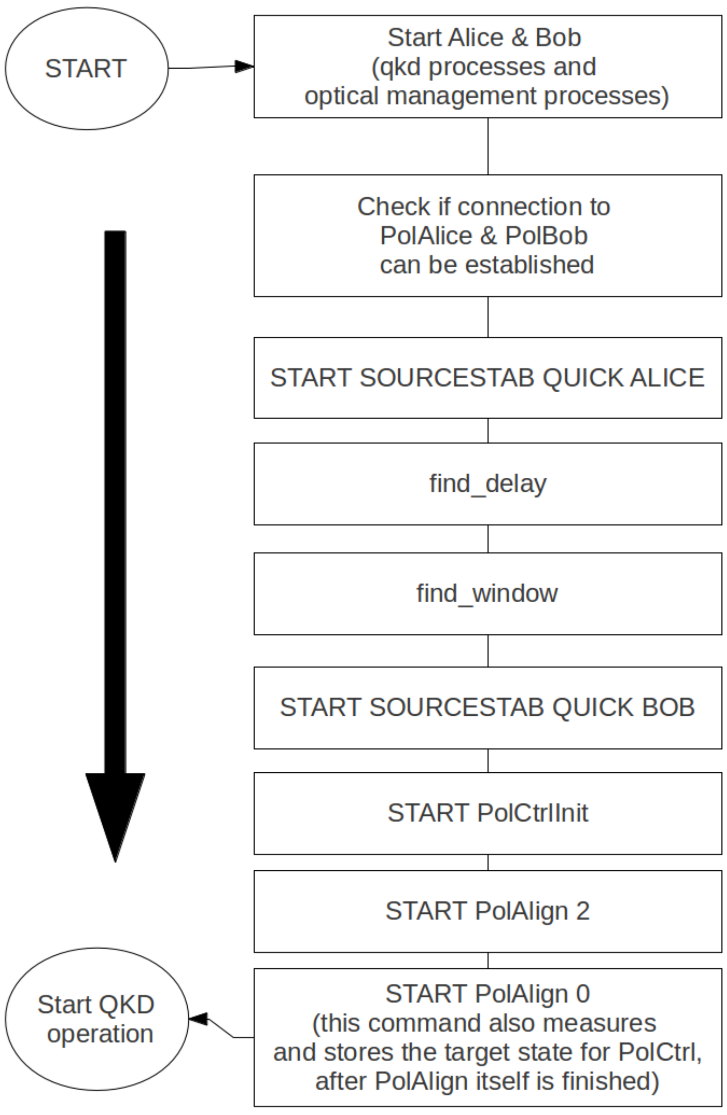
START
Start Alice & Bob (qkd processes and optical management processes)
Check if connection to PolAlice & PolBob can be established
START SOURCESTAB QUICK ALICE
find_delay
find_window
START SOURCESTAB QUICK BOB
START PolCtrlInit
START PolAlign 2
START PolAlign 0 (this command also measures and stores the target state for PolCtrl, after PolAlign itself is finished)
Start QKD operation

Figure 6.3.: Optical management start-up sequence

The normal operation sequence (after start-up is completed):

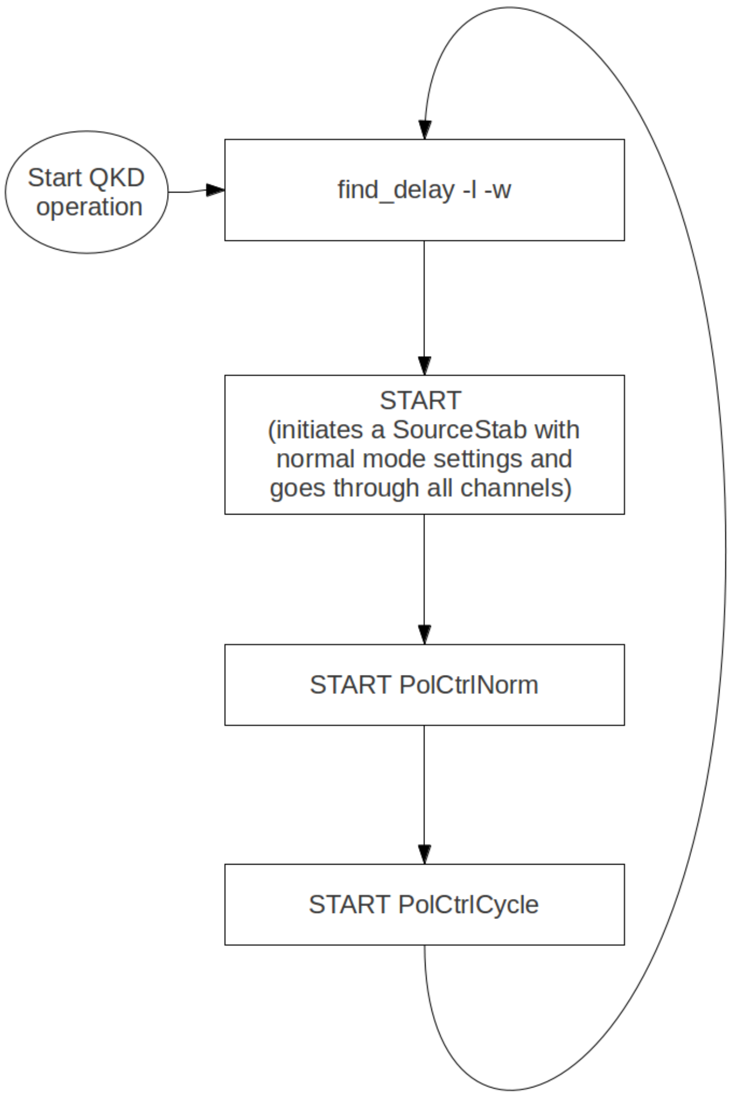
Start QKD operation
find_delay -l -w
START
(initiates a SourceStab with normal mode settings and goes through all channels)
START PolCtrlNorm
START PolCtrlCycle

Figure 6.4.: Optical management normal operation (without error handling) sequence

Furthermore, in the background we always use "TEMP" after each procedure that was initiated by a command (during the normal operation sequence) to check if the current crystal temperature inside a certain range (this range can be specified through configuration parameters which are described in detail in chapter 6.5).

If the crystal temperature is outside this range, a warning message is written into the log file. This could be improved in the future to e.g. send an e-mail too.

With these sequences and the earlier descriptions of the individual procedures (see chapter 6.2) it should be clear what the system does during the start up and during the normal operation.

However, it doesn't include the error handling yet.
Error handling is possible AFTER a successful start up and if needed interferes with the normal operation of the system. After a successful error handling the normal operation cycle always returns to the first item "find_delay -l -w" (further details regarding error handling in chapter 6.4).

6.4. Error handling

This is probably one of the most important features of the optical management system as it is necessary to ensure stability and fast response in case of low or zero keyrates.

First of all, it is nessary to mention that various previous error handling procedures (described here [7, Chapter 6.5.3 Page 55]) were not availble or usable during SECOQC. Actually, only two errors were handled and even those were only handled partially.

I will now explain all the errors that the management module handles and how they are handled and during this it will also be explained in what way or if at all the system handle those errors during SECOQC.

This incomplete error handling during SECOQC was probably because of time pressure and because it wasn't so easy to interact between the processes like now.

In order to handle the error, the optic_ctrl process first determines if there is an error and what kind of action should be used.

optic_ctrl then informs optic_master what type of action should be used. It also keeps this information stored because for certain errors it is possible that the first action doesn't resolve the error and we have to change what kind of action is executed
In order to do so it is necessary to know what type of action was executed immediately before.

Furthermore, the information sent to optic_master about the required action is kept there until optic_master is finished with the action. It then resets this information.
This is necessary to let optic_ctrl know that it shouldn't check for (and therefore inform optic_master about) further errors.
We have to do this to avoid confusion between error handling. Not to mention that it doesn't make sense to react to an error that might not actually exist anymore AFTER the error handling is finished.

Error handling procedures aren't supposed to interrupted. This is because most of those procedures would leave the system unusable if interrupted in between or it is finished in a few seconds that it doesn't really matter.

As mentioned all errors if an error handling action is in progress are discarded because they might be fixed after the procedure anyway. If it isn't fixed afterwards, we will find out about it anyway and react to it accordingly.

However, there is an exception to this and that is if we are notified about a possible "Makarov attack". In that case we immediately forward the error to optic_master, even if there is currently another error handling in progress. At optic_master we then either act immediately on this or wait until the system is in a situation where we can act again (normally it is either soon afterwards or the error handling has stopped the key production anyway).
This is done because such an attack compromises the key integrity and we therefore have to ensure no new keys are being produced.
Furthermore, we are only informed about an attack and don't check periodically if an attack has occurred.

The same goes about the information that a "Makarov attack" has ended.
This is another difference to normal error handling. In contrast to the normal error handling where optic_master informs optic_ctrl that the error handling was completed, it is different here because optic_master will stay in an error state until optic_ctrl informs it that the Makarov attack has stopped by resetting the error information.

Now we will take a look at the different error handling actions (what they do or what procedures they invoke and why) that optic_ctrl forwards to optic_master and afterwards we will explore under what conditions those actions are invoked.

6.4.1. Error handling actions

This chapter describes how optic_master reacts to the error handling commands from optic_ctrl. Most of those actions invoke various procedures described in chapter 6.2.
When those actions are invoked will be explained at chapter 6.4.2 and it will also explain (indirectly) why those names have been chosen.

1. **OPTIC_CMD_DRIFT (action for QBER Drift):**
   We normally have to use a QBER Drift action, if there are polarisation changes (e.g. because of temperature changes) because of the fibres inside Bob (e.g. because of temperature changes) after the 95:5 beam splitter that seperates the quantum channel and the polarisation control.
   If that is the case the only thing that can be done is shown in the Figure 6.5.
   Basically we are re-aligning the states. Of course, because it takes quite some time (several minutes) and the key generation is stopped during its execution, this type of action should be avoided, if possible.

2. **OPTIC_CMD_DIST (action for fibre distortion):**
   A fibre distortion action is used if there is a sudden big increase in the QBER which is likely to be caused by some distortion of the fibre between the devices.
   If that is the case we are trying to fix it by what is shown in the Figure 6.6.
   Basically we are trying if the polarisation control can fix this error.

3. **OPTIC_CMD_50 (action for 50 % error):**
   This action is executed if a trigger pulse from Alice to Bob is lost, and therefore, causes the key data between Alice and Bob to be unsynchronized.
   The only way to fix this is shown in Figure 6.7 and it is necessary to do so because we have to empty the buffers of the QKD processes.

4. **OPTIC_CMD_MAKAROV (action for a Makarov attack):**
   It just stops QKD operation and all qkd stack processes because we cannot accept any keys that would be produced during such an attack and should get rid of key material (that is stored in the qkd stack processes buffer) because it might already be slightly influenced by the Makarov attack.

   If a Makarov attack is stopped we are informed through the action handler being modified to a no error state (that would be OPTIC_CMD_OK).
   The system then restarts/resumes the qkd stack processes and QKD operation as the data between the devices is no longer compromised.

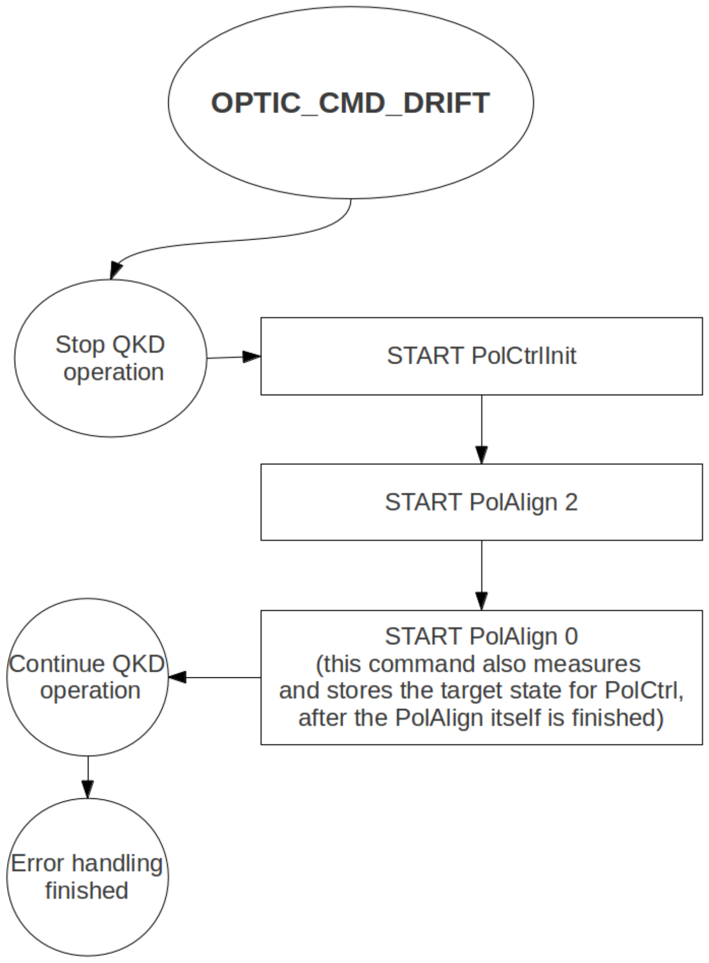
OPTIC_CMD_DRIFT
Stop QKD operation
START PolCtrlInit
START PolAlign 2
START PolAlign 0 (this command also measures and stores the target state for PolCtrl, after the PolAlign itself is finished)
Continue QKD operation
Error handling finished

Figure 6.5.: QBER Drift action (OPTIC_CMD_DRIFT) handling

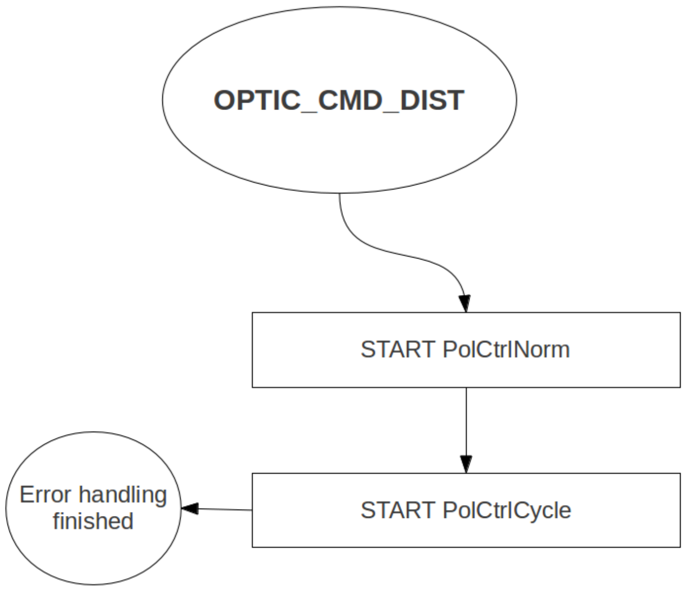

Figure 6.6.: Fibre distortion action (OPTIC_CMD_DIST) handling

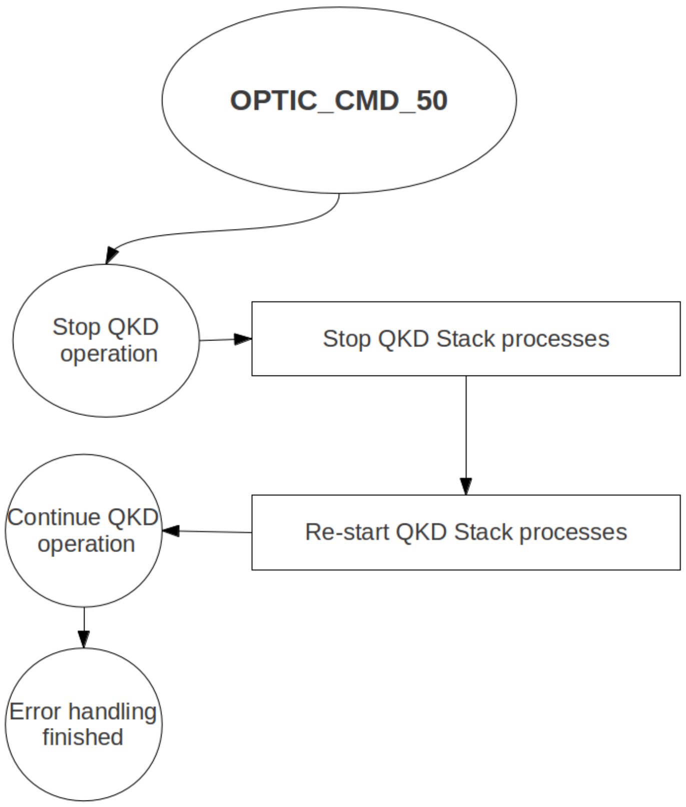

Figure 6.7.: 50 % error action (OPTIC_CMD_50) handling

6.4.2. Conditions for error handling actions

The system reacts to three (four if we include "Makarov attack") different type of errors which invoke the actions described in chapter 6.4.1 under certain changes of the QBER.

The error type is defined by the action that successfully counteracts an error. Under certain circumstances this error type can also be determined by the condition for the error. Therefore, the error type will be used to describe the conditions.
However, the condition might not always describe the correct error type though. It can happen that further action than the default action is required to get rid of the error. This will be explained for the individual conditions (all of them depend on the QBER).
If further action is required that action is determined by the original error condition and takes into account that the error was still after the error handling.
Of course, it is wise to wait one QBER/key value to be sure that the condition isn't just met again because the current QBER value/key still includes data from before the error was fixed. Actually the amount of QBER values/keys that the system waits can be specified through a parameter (see chapter 6.5) but the default value is "1" and it is also the value that makes the

most sense.

1. **Fibre distortion:**

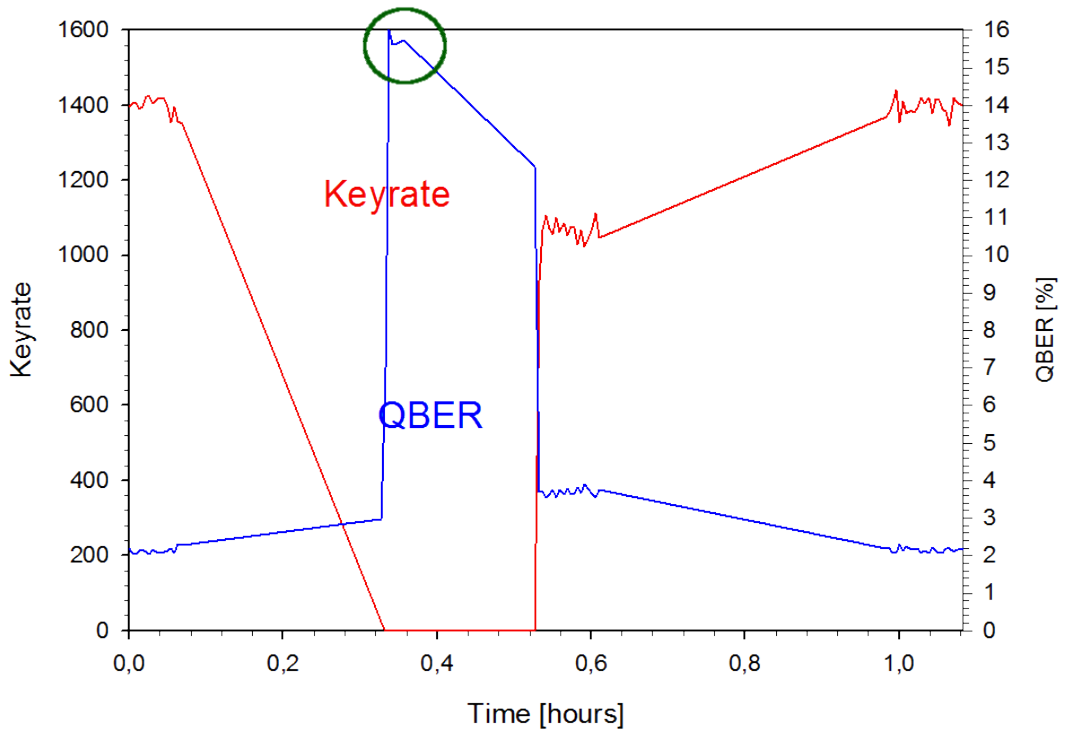

Figure 6.8.: Condition for a fibre distortion

What happens here is that the QBER suddenly increases to quite a high value (marked by the green circle in Figure 6.8).

So the condition is simply to check if the current QBER value is higher than a defined limit. If it is then system has to take action for this type of error.
The defined limit is a parameter and can be changed (see chapter 6.5).

This type of error might not be resolved through one action which means that - as described above - if this condition is still met after the error handling action is finished (and we waited a certain amount of QBER values/keys) we will do a different action.

The order of the executed actions is:

a) **OPTIC_CMD_DIST:**
First we are trying to handle it with this type of error handling because this one is finished in a few seconds and doesnt really disturb QKD.

b) **OPTIC_CMD_DRIFT:**
If the original action didn't help, the only other way to fix this is by invoking this action.
However, as described in chapter 6.4.1 this action stops QKD and takes quite some time (several minutes) and should therefore be avoided. Of course, under certain circumstances (like this one) this is the only option left.

In the old optical management system used during SECOQC this error condition only invoked OPTIC_CMD_DIST. There was no further error handling, if this didn't work.

2. **QBER drift:**

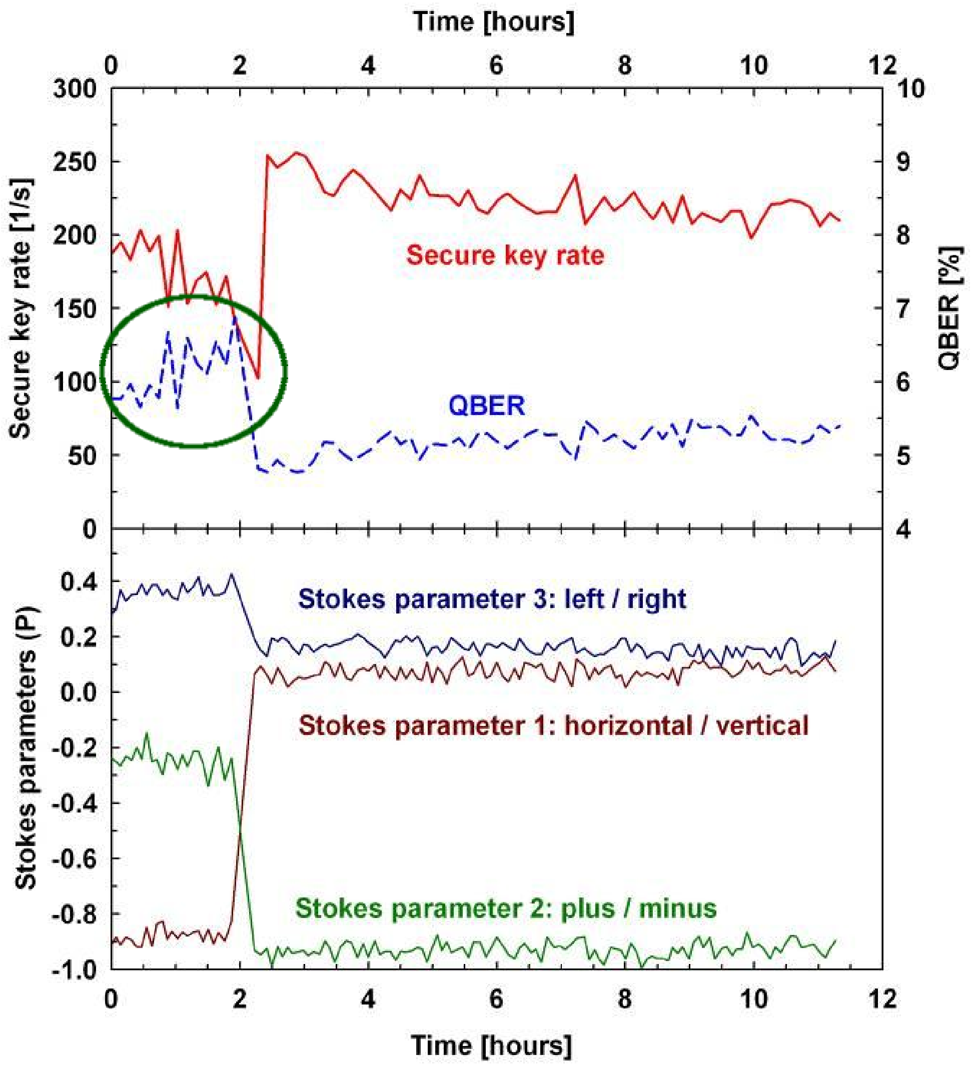

Figure 6.9.: Condition for a QBER drift, from "Field test of quantum key distribution in the Tokyo QKD Network" [1, Page 10405 Fig. 17]

The condition is basically described through the section that is marked by the green circle in Figure 6.9.
It means that there is a slow but steady increase in the QBER.

Actually, the full condition is that after a StateAlign we generate a reference QBER by creating a mean value of a certain amount of QBER values (this amount can be specified through a parameter, see chapter 6.5).
The calculation for the reference value is quite different to the old optical management system used during SECOQC. In the old management system the reference value was only

calculated once after the system start up. There was no refresh after a successful QBER Drift correction (StateAlign) which, of course, can cause the reference value to be quite different from the one before.

However, not all values are included in the mean value calculation for the reference QBER and the mean value (in the new optical management; for the old optical management used during SECOQC all values were included in the calculation). If the values are too high they are discarded and we wait for the next one.
Too high values are for the mean QBER calculation if they are higher than the limit for a fibre distortion and for the reference value it is that limit lowered by the maximum allowed difference for a the reference value and the mean value.

The reason why we ignore such a high values for the reference value is simple that the whole QBER drift error handling doesn't work well if the reference value is already too high.
And the reason why we ignore such high values for the mean value is because those values are already handled through the fibre distortion and furthermore they would give a huge influence to the mean value, even if they only occur once.

After the reference value is successfully calculated we are then always creating a new mean value in the background and compare it to the reference value. If the difference between the reference value and the current mean value is too big (maximum difference is set through a parameter, see chapter 6.5) we run into this error.

The used action for this error is just:
OPTIC_CMD_DRIFT

3. **50% error:**
   For this condition to be reached the QBER jumps to a value of about 50%.
   This is most likely caused by losing a trigger pulse and therefore the key data at Alice and at Bob is out-of-sync which results in such an error value.

   In our case we are using two values to check if this is a 50% error. It has to be above the first value and below the second value. The default values are both near 50%. Those two values are parameters (see chapter 6.5).

   This error condition (and therefore its error handling) didn't exist in the old optical managment system during SECOQC.

If we run into this condition the actions used are:

a) **OPTIC_CMD_DIST:**
Though it is a so called 50% error type condition, we are not starting with a OPTIC_CMD_50 because this error could also be a fibre distortion - even if it is unlikely - which is fixed in a few seconds by using OPTIC_CMD_DIST.
The reason why we use this action first is that a OPTIC_CMD_50 stops qkd and all qkd-processes which clears all key material that is currently stored in the buffer of the qkd-processes. This means that we loose quite a lot of key data and therefore also the time without new keys increases (especially now that we added the qkd_queue process). So it is better to first try OPTIC_CMD_DIST.

b) **OPTIC_CMD_50:**
If the fibre distortion didn't work we have no choice but do the real 50% error handling.

c) **OPTIC_CMD_DRIFT:**
If all of this didn't work, we only have one more choice and that is to invoke the OPTIC_CMD_DRIFT. As already mentioned this action takes several minutes and is therefore the last action we try.
This is mostly done to make sure that we tried every possible option to fix this error because we won't get any new keys as long as this error persists. However, it actually should never be required to get this far but it isn't impossible.

4. **Makarov attack:**
As already mentioned in chapter 6.4.1 this error is invoked by getting an information from PolBob that a Makarov attack occured.
This error will only be resolved by another message from PolBob that informs as that the Makarov attack has seized.
In order to find out what exactly happens if such an attack occurs look at OPTIC_CMD_MAKAROV of chapter 6.4.1.

This error condition (and therefore its error handling) didn't exist in the old optic management system used during SECOQC.

6.5. Parameters for the processes (optical management)

In this chapter we will go through all the parameters that can be specified in the individual processes for the optical management.
We will further distinguish between the parameters the processes get and the parameters the processes export.
Parameters are values that are exchanged through the new QKD process management system (by using DBus, see [5]), see chapter 4.) for detals.
They are mostly used for configuration purposes but not solely.

For the tables below the column "Name" represents the parameter name to use which can be set over the control program for the new qkd process management system or in case of exported values to check on them through DBus directly.
"Default" means the used default value for this parameter, if it isn't set/changed manually and "Initial" means the initial value that is used for an exported parameter until the first change of that value by a process.

1. **optic_master:**
   This processes only gets parameters, it doesn't export any.

Name	Default	Description
mode	0	This value specifies what type of start up and operation the processes shall do (also check chapters 6.3 and 6.4). Possible values are: 0 ... a complete start up sequence with normal operation (normal operation cycle and error handling) afterwards 1 ... a complete start up sequence but then enters an idle operation (no normal operation cycle and no error handling) 2 ... a quick start up sequence which means that we skip SourceStab and StateAlign but we do find_delay and find_window; afterwards it runs in normal operation 3 ... no start up sequence, just starting QKD operation and then run in normal operation 4 ... no start up sequence, just starting QKD operation and then run in idle operation

second-monitor-application	false	This option specifies if we should not only send the information about the commands optic_master has issued not only to the main monitoring application but also to a second one that is running on Node-Alice (where also optic_master runs) and can be used by some tools to log this status (like e.g. optic_master issued the command to do a PolCtrl-Cycle) with their internal timing scheme. Possible values are: false ... NO information is sent to a second monitoring application true ... also sends the information to a second monitoring application
delay-time	0	During the normal operation every time before "find_delay -l -w" (so basically after a cycle is completed) we wait for the amount of seconds which are specified by this value. So the accepted values are greater than or equal zero. The default value is set to zero as there is no actual need to wait between the cycles.
cycle-time-norm	0	time between the PolCtrl-Cycles during normal operation sequence. Therefore its value should be greater than or equal zero. As we are now only doing one PolCtrl-Cylce (which takes care of doing the operation more than once, if necessary) we are only using one PolCtrl-Cycle and therefore there isn't really any time required between those cycles and therefore the default value is zero.
cycle-time-corr	0	The same as the cycle-time-norm parameter, however, instead of during the normal operation sequence this is used if we handle an OPTIC_CMD_DIST action.

cycle-counts-norm	1	This parameter specifies how many PolCtrl-Cycles should be done during the normal operation sequence. With various changes to the PolCtrl-Cycle - especially, that if needed we do more than one cycle/operation anyway - it isn't necessary anymore to do this more than once. Actually, it would just lengthen the time until we reach other actions of the normal operation cycle which might be needed more.
cycle-counts-corr	1	Same as cycle-counts-norm parameter, however, this one is used during an OPTIC_CMD_DIST action.
crystal-temp-limit-lower	3000.0	This parameter specifies the lower limit for the temperature check during normal operation. If the crystal temperature falls below (or is equal to) this value there will be a warning in the log file.
crystal-temp-limit-upper	3100.0	This parameter specifies the upper limit for the temperature check during normal operation. If the crystal temperature rises above (or is equal to) this value there will be a warning in the log file.
unix-socket		This is used to specify a path to a ”pipe” which is required for the communication with ”qdev-q3pdev-bridge” and in this case to forward information over authenticated channels to Bob. There is no default value for this parameter. It must be specified.
info-socket		Basically the same as unix-socket, however, it is used for information messages that go only to ”qdev-q3pdev-bridge” and not further.
verbosity	0	This parameter is used to determine how much information is written into the log file. With a lower value less information is written into the log file. 0 is the lowest value.
initiator	true	This parameter should be set to true for this program. It is actually a parameter that exists for all qkd processes per default and it is normally set to true on processes that are started at Alice.

If this program is started through the new qkd process management, it could be done like this (this is only an example, for instance various paths could be different etc.):

```
qkd-ctl pipe-create optic-master
qkd-ctl pipe optic-master clear
qkd-ctl process-create optic_master /usr/local/bin/optic_master

qkd-ctl process optic_master prop unix-socket=/tmp/optic1_dev1.sock \
                                  info-socket=/tmp/optic1_info1.sock \
                                  verbosity=3 \
                                  mode=0 \
                                  second-monitor-application=false \
                                  delay-time=0 \
                                  cycle-time-norm=0 \
                                  cycle-time-corr=0 \
                                  cycle-counts-norm=1 \
                                  cycle-counts-corr=1 \
                                  crystal-temp-limit-lower=3000 \
                                  crystal-temp-limit-upper=3100 \
                                  initiator=true

qkd-ctl pipe optic-master append optic_master
qkd-ctl pipe optic-master start
```

2. **optic_ctrl:**
First the parameters the process receives:

Name	**Default**	**Description**
sample-counts	12	This parameter defines how many QBER values are used for mean value and reference value calculation for the QBER drift condition (see chapter 6.4.2 and check condition "QBER drift").
qber50-lower-bound	4750	This value is used to set the lower bound for the 50% error condition (see chapter 6.4.2 and check condition "50% error"). It is the QBER value (in %) times 100, so this default value would mean a QBER of 47.50 %.

qber50-upper-bound	5250	This value is used to set the upper bound for the 50% error condition (see chapter 6.4.2 and check condition "50% error"). It is again the QBER value (in %) times 100, so this default value would mean a QBER of 52.50 %.
qber-limit	800	This value is used the QBER limit for the fibre distortion condition (see chapter 6.4.2 and check condition "Fibre distortion"). It is again the QBER value (in %) times 100, so this default value would mean a QBER of 8.00 %.
qber-drift	100	This parameter defines the allowed difference of the QBER value to the reference value for the QBER drift condition (see chapter 6.4.2 and check condition "QBER drift"). It is again the QBER value (in %) times 100, so this default value would mean a QBER of 1.00 %.
det-0counts	0	If we get this amount of zero counts consecutively at one detector, we will report in the log file that there was an error with the detector. However, if this value is set to 0 (as the default value) this check is disabled and there won't be any detector failure reports to the log file.
wait-keys-after-fix	1	This amount of keys/QBER values will be waited after an error was fixed before the error handling is resumed (as mentioned and described in chapter 6.4.2).

And now the parameters that the process exports:

Name	**Initial**	**Description**
error	0	This parameter described the current error handling action (see chapter 6.4.1 to find what the error handling actions mentioned below do) that optic_ctrl tells optic_master to do. Each of these error handling actions has a corresponding number (and because internally these numbers are also used for other actions that are not error related, they are not in ascending order): 0 ... OPTIC_CMD_OK (which means that there currently is no error handling action to do as everything is working correctly) 17... OPTIC_CMD_50 18... OPTIC_CMD_DRIFT 19... OPTIC_CMD_DIST 20... OPTIC_CMD_MAKAROV
mean-qber	0	current mean value of QBER values (in % times 100) that is internally used to compare to the reference value for the QBER drift condition
reference-qber	0	the calculated reference QBER value (in % times 100) for the QBER drift condition checks
current-qber	0	the current received QBER value (in % times 100)
unix-socket		This is used to specify a path to a "pipe" which is required for the communication with "qdev-q3pdev-bridge" and in this case to forward information over authenticated channels to Bob. There is no default value for this parameter. It must be specified.
info-socket		Basically the same as unix-socket, however, it is used for information messages that go only to "qdev-q3pdev-bridge" and not further.
verbosity	0	This parameter is used to determine how much information is written into the log file. With a lower value less information is written into the log file. 0 is the lowest value.
initiator	true	This parameter should be set to true for this program. It is actually a parameter that exists for all qkd processes per default and it is normally set to true on processes that are started at Alice.

If this program is started through the new qkd process management, it could be done like this (this is only an example, for instance various paths could be different etc.):

```
qkd-ctl pipe-create optic-ctrl
qkd-ctl pipe optic-ctrl clear
qkd-ctl process-create optic_ctrl /usr/local/bin/optic_ctrl

qkd-ctl process optic_ctrl prop unix-socket=/tmp/optic1_dev0.sock \
                                info-socket=/tmp/optic1_info0.sock \
                                verbosity=3 \
                                sample-counts=12 \
                                qber50-lower-bound=4750 \
                                qber50-upper-bound=5250 \
                                qber-limit=800 \
                                qber-drift=100 \
                                det-0counts=0 \
                                wait-keys-after-fix=1 \
                                initiator=true

qkd-ctl pipe optic-ctrl append optic_ctrl
qkd-ctl pipe optic-ctrl start
```

3. **optic_slave & optic_tunnel:** They both only get parameters and they both have the same type of parameters:

Name	**Default**	**Description**
unix-socket		This is used to specify a path to a "pipe" which is required for the communication with "qdev-q3pdev-bridge" and in this case to forward information over authenticated channels to Alice. There is no default value for this parameter. It must be specified.
info-socket		Basically the same as unix-socket, however, it is used for information messages that go only to "qdev-q3pdev-bridge" and not further.
verbosity	0	This parameter is used to determine how much information is written into the log file. With a lower value less information is written into the log file. 0 is the lowest value.

If you want to start these programs through the new qkd process management, it could be done like this (this is only an example, for instance various paths could be different etc.):

For **optic_slave:**

```
qkd-ctl pipe-create optic-slave
qkd-ctl pipe optic-slave clear
qkd-ctl process-create optic_slave /usr/local/bin/optic_slave

qkd-ctl process optic_slave prop unix-socket=/tmp/optic0_dev1.sock \
                                 info-socket=/tmp/optic0_info1.sock \
                                 verbosity=3 \

qkd-ctl pipe optic-slave append optic_slave
qkd-ctl pipe optic-slave start
```

For **optic_tunnel:**

```
qkd-ctl pipe-create optic-tunnel
qkd-ctl pipe optic-tunnel clear
qkd-ctl process-create optic_tunnel /usr/local/bin/optic_tunnel

qkd-ctl process optic_tunnel prop unix-socket=/tmp/optic0_dev0.sock \
                                  info-socket=/tmp/optic0_info0.sock \
                                  verbosity=3 \

qkd-ctl pipe optic-tunnel append optic_tunnel
qkd-ctl pipe optic-tunnel start
```

7.). UQCC 2010 measurements & statistics

We did some tests in the laboratory and then, of course, there was the demonstration of the Tokyo QKD network during UQCC 2010 (with fibre length for QKD transmission of approximately 1km).
In the laboratory we only did long time frame tests with a short fibre (similar length to the Tokyo QKD network demonstration) because of the bad crystals (see further down) and because of the distance at the Tokyo QKD network demonstration to have a similar testing environment.

It was already noticeable during the laboratory tests that the keyrate was lower as in SECOQC but it was even worse in UQCC 2010.
The problem seems to be the crystals. They are now causing light dispersion and depending on the incoming direction and position of the light, the resulting keyrate is better or worse. This different direction and position is the main difference between UQCC 2010 and the laboratory tests before.
In UQCC 2010 we did not want to risk losing the keyrate altogether before the demonstration which is why we just optimized the existing keyrate and did not try to find a different and better spot.

The reasons that the crystals deteriorated is most likely caused by leaking heat paste that got hit by the laser beam when the system is turned on. This would at least cause the coating of the crystals to be damaged.

To further demonstrate this difference, I will first quote a figure which shows the keyrate and QBER value of the laboratory tests during SECOQC from "A Fully Automated Quantum Cryptography System - Based on Entanglement for Optical Fibre Networks (Diploma Thesis)" [7, Figure 7.5a) Page 63] and one that shows the keyrate and the QBER value during the SECOQC demonstration - where the system was used over a 16km distance - from [7, Figure 8.10 Page 73].

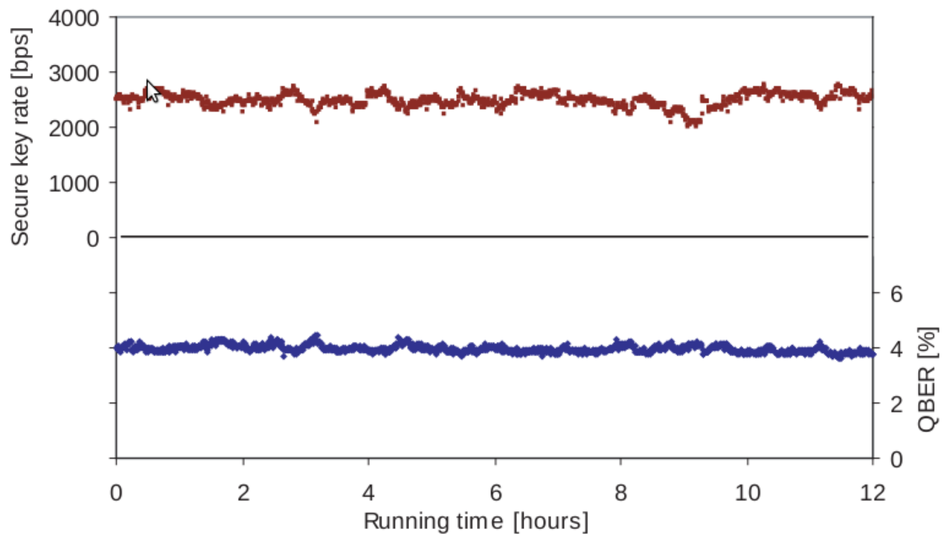

Figure 7.1.: QBER and keyrate from **laboratory tests before SECOQC**, quoted from [7, Figure 7.5a) Page 63]

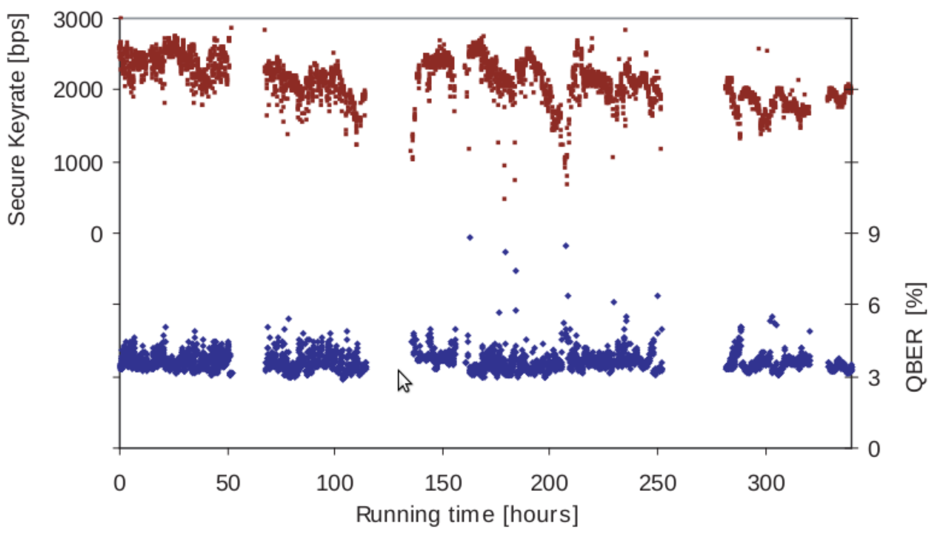

Figure 7.2.: QBER and keyrate from the SECOQC demonstration, quoted from [7, Figure 8.10 Page 73]

In contrast to those here we have the lab results prior UQCC 2010:

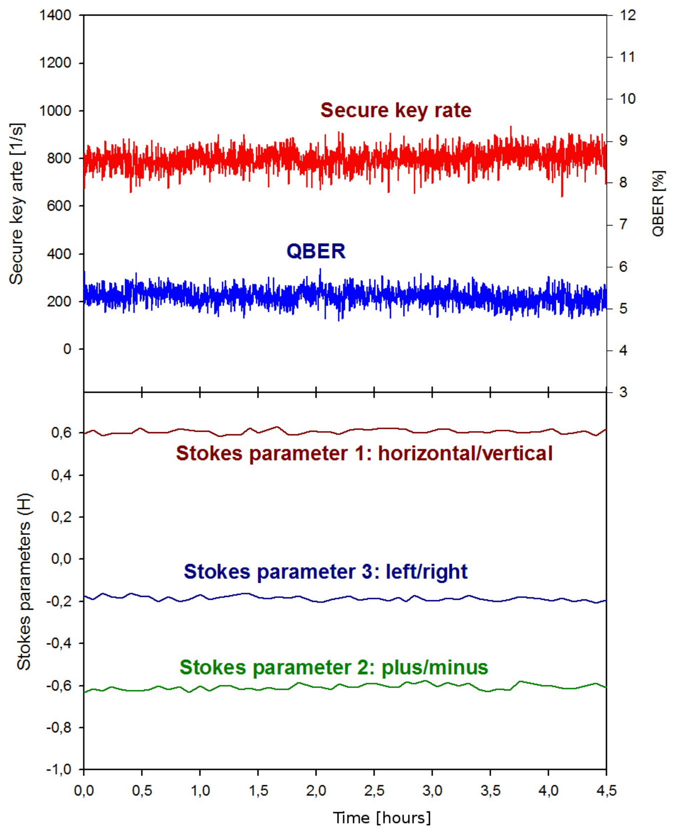

Figure 7.3.: QBER and keyrate from **laboratory tests before UQCC 2010**

And here the results during UQCC 2010:

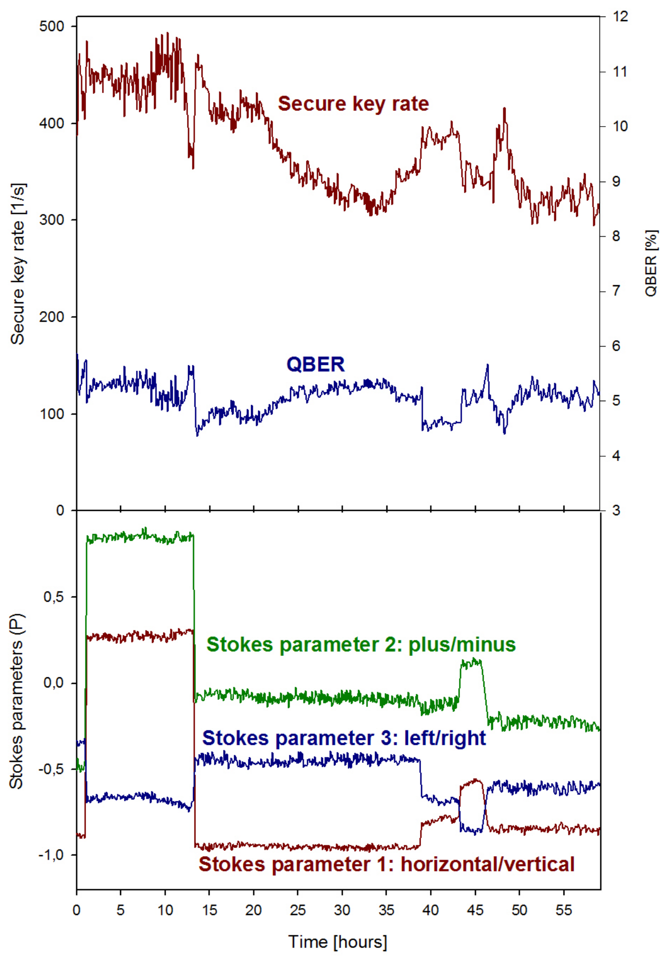

Figure 7.4.: QBER and keyrate from **UQCC 2010's Tokyo QKD network demonstration**

These figures show that even though the keyrate is low the system is still stable. During the Tokyo QKD network demonstration it was completely stable for about 12 to 13 hours. Then something caused the current state to change. Most likely a larger shift of the quantum channel fibre. This can be seen because of the huge change in the Stoke's parameters (see Figure 7.4). However, because of the polarization control the whole system was stable stable although the keyrate deterorated a bit.

It is also feasible that the changes happened because now someone entered the laboratory and switched on the lights (as we did not use the system cover during Tokyo QKD network demonstration to be able to make adjustments to the entanglement source before the demonstration and to show the inside of the system during the presentation).
Also in Japan fibres are not in the ground but mostly in the air, therefore, fibre "movement" is to be expected.

However, the general decrease (after those completely stable 12 to 13 hours) in the keyrate and QBER does not seem to be caused by the polarization control but more likely because of the crystal quality. The reason is that through the source stabilization the system tries to keep the photon countrates constant, however, because of the dispersion in the crystals it is possible that the source stabilization let the light enter in a way that enhanced the dispersion.

Therefore, it can happen that the source stabilization (in combination with temperature drifts) changes the position and direction in which the light goes through the crystals. As already mentioned, the crystals are quite deteriorated and therefore the direction and/or position of the light that passes through the crystals can result in a higher dispersion and therefore lower keyrate. In order to get back to the maximum the procedure would likely need to pass through a local minimum of the keyrate which, of course, it does not do during the automatic procedure. Therefore, with good crystals this problem would not have occured.

However, there was no 0 keyrate during the whole run it means the system was stable for the whole time (over 2 days) without any state alignment necessary (see chapter 6.4) and was able to be completely stable because of the polarization control alone.

Not to mention that the keyrate was lower than during SECOQC the whole time and it still was stable.
This proves that the changes to the optical management (see chapter 6.)) which also include better error handling (upon keyrate and QBER changes) and the adding of the in-line polarizers (see chapter 3.2.3) helped to get the system more stable even under worse conditions.

8.). Makarov Attacks

I am now going to explain a bit about three attacks on the detectors (that are relevant for our system because of the optical parts described in chapter 3.)) found and published by the group of Vadim Makarov.
Those three attacks are:
"Controlling passively quenched single photon detectors by bright light" - see also [15] - (such an attack will be called "Bright light" during the rest of this chapter), "Thermal blinding of gated detectors in quantum cryptography" - see also [16] - (such an attack will be called "Thermal blinding" during the rest of this chapter) and "After-gate attack on a quantum cryptosystem" - see also [17] - (such an attack will be called "After-gate attack" during the rest of this chapter).
Our system has possibilities to counteract those attacks which will also be explained.
The same group also proposed a way to secure the detectors against such attacks in "Secured gated detection scheme for quantum cryptography" [19].

8.1. Bright light

This section describes how to control passively quenched single photon detectors by bright light (detailled description can be found in [18] and [15]). The main point is that single photon detectors based on passively quenched avalanche photodiodes (as used in our QKD system) can be blinded to single photon detection using high intensity cw illumination. Normally a APD is reversed-biased above the breakdown voltage, which causes single photons to induce a large current (avalanche) through the APD. However, by using high intensity cw (continous-wave) illumination the Bias voltage drops below the breakdown voltage and therefore the detector is not capable of detecting single photons anymore. Once the detectors are blinded they can easily be controlled by classical laser pulses which are super-imposed over the blinding (cw) signal.

Therefore it is possible to have an interceptor (Eve) of the quantum channel which remains undetected by Alice and Bob as this one can forward its own results to Bob.

Our system uses the 5% fraction of the light from the 95:5 beam splitter (see chapter 3.)) on Bob's side which goes in the polarimeter to detect bright light pulses (as the voltages for the polarimeter outputs will be too "high") and if this happens to stop QKD key generation (see also chapter 6.4).

8.2. Thermal blinding

The thermal blinding principle is similar to the one of the bright light attack (see chapter 8.1), however, instead of decreasing the reverse Bias voltage this attack increases the breakdown voltage. cw light can be used to dissipate the power in the APD to thermal heat which may increase the APD's temperature. As the breakdown voltage increases with higher temperature this can cause the breakdown voltage (after the APD was "heated"/illuminated long enough) to increase above the applied Bias voltage (further details can be found in [16]).
Therefore, the detector would become blinded and can again be manipulated by super-imposing classical laser pulses over the bright light signal.

According to the talk I had at UQCC2010 with one of Makarov's team members the detectors used by our system cannot be blinded this way because they would shut down of excessive heat and wait for cooldown.
Furthermore, this attack should also be detectable the same way as the bright light Makarov attack (see chapter 8.1).

8.3. After-gate attack

APDs are only biased above the breakdown voltage during the detection gate which means that before and after the reverse Bias voltage is lower than the breakdown voltage and therefore the APD is in the linear (classical) detection mode during that time.

Various systems/detectors accept "single photon click events" after and before the detection gate which can be exploited. However, before the gate will result in a higher QBER (becaue of large after-pulses, see [17]) and can therefore not be used in an attack.
On the contrary, it is possible to exploid the linear mode after the gate for an attack by using bright laser pulses during that time to produce fake clicks.

As our system also uses a software window for allowed detection clicks (see find_window of chapter 6.2 and the window size can be set through parameters) it should be possible to avoid this kind of attack by using a small enough software window.

9.). Conclusion

This work demonstrates how an entanglement based quantum cryptography system can be achieved and that it is stable to be used in combination with other quantum cryptography systems in the UQCC 2010 Tokyo QKD network demonstration.

Other quantum cryptography systems that were used in that demonstration are outlined in the paper
"Field test of quantum key distribution in the Tokyo QKD Network", Optics Express, Vol. 19, Issue 11, pp.10387-10409 (2011) [1].

Although a similar demonstration was already done during SECOQC, this work also explains points that have been improved like a new optical management system which was necessary to have good error handling (e.g. if the QBER increases too much) and changes in the optical layout of the system.
It also shows that because of the changes (management and optical layout) the system is more stable than before, even with a less keyrate because of bad crystals.

Because our system uses entanglement it would allow further improvements than most other QKD systems.
For instance, through measuring the Bell states it would be possible to create a system-independant QKD system.

Furthermore, this work shows that although quantum cryptography is already a well researched field, there is still room for further reasearch and improvement.

Mostly, in the direction of side channel attacks like the various
Makarov attacks (see chapter 8.)) and, of course, higher keyrates which mostly requires better detectors. For instance, the Toshiba team has already achieved some improvments in that direction as they have shown during their UQCC 2010 Tokyo QKD network demonstration (further details see [1]).

Bibliography

[1] M. Sasaki, M. Fujiwara et al.: Field test of quantum key distribution in the Tokyo QKD Network, Optics Express, Vol. 19, Issue 11, pp.10387-10409 (2011),
Web: http://www.opticsinfobase.org/oe/abstract.cfm?URI=oe-19-11-10387

[2] Nicolas Gisin, Gregoire Ribordy, et al.: Quantum cryptography, Reviews of Modern Physics, Vol. 74, pp.145-195

[3] Peter W. Shor, Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer, arXiv:quant-ph/9508027v2

[4] Michael A. Nielsen, Optical quantum computation using cluster states , arXiv:quant-ph/0402005v1

[5] http://www.freedesktop.org/wiki/Software/dbus

[6] https://sqt.ait.ac.at/software/wiki/qkd-stack/

[7] Alexander Treiber: Diploma Thesis - A Fully Automated Quantum Cryptography System - Based on Entanglement for Optical Fibre Networks, 24. 08. 2009

[8] Alexander Treiber: User Manual für das Entangled Photons Quantum Cryptography System; Spezifikation v2.5; 21. 01. 2010; only available internal

[9] Alexander Treiber: Management-Modul für das Entangled Photons Quantum Key Distribution System; Spezifikation v2.1; 17. 11. 2009; only available internal

[10] Alexander Treiber, Andreas Poppe et al.: A Fully automated entanglement-based quantum cryptography system for telecom fibres, New Journal of Physics 11 (2009), 045013

[11] Rodney Van Meter, Thaddeus D. Ladd, W.J. Munro, Kae Nemoto: System Design for a Long-Line Quantum Repeater, arXiv:0705.4128v2

[12] Lluis Masanes, Stefano Pironio, Antonio Acin: Secure device-independent quantum key distribution with causally independent measurement devices, arXiv:1009.1567v2

[13] Hannes Hubel , Michael R. Vanner et al.: High-fidelity transmission of polarization encoded qubits from an entangled source over 100 km of fiber, Optics Express, Vol. 15, Issue 12, pp. 7853-7862 (2007)

[14] Thorben Kelling: Towards a HandsOff Plug&Play Quantum Key Distribution System with Entangled Photons for a Quantum Network. Master's thesis, University of Heidelberg, 2008

[15] V. Makarov: Controlling passively quenched single photon detectors by bright light, New Journal of Physics 11, 065003 (2009).

[16] Lars Lydersen, Carlos Wiechers, Christoffer Wittmann, Dominique Elser, Johannes Skaar and Vadim Makarov: Thermal blinding of gated detectors in quantum cryptography, Optics Express, Vol. 18 Issue 26, pp.27938-27954 (2010)

[17] C Wiechers, L Lydersen, C Wittmann, D Elser, J Skaar, Ch Marquardt, V Makarov and G Leuchs: After-gate attack on a quantum cryptosystem, New Journal of Physics 13, 013043 (2011).

[18] Lars Lydersen, Carlos Wiechers, Christoffer Wittmann, Dominique Elser, Johannes Skaar and Vadim Makarov: Hacking commercial quantum cryptography systems by tailored bright illumination, Nature Photonics, Volume 4 (2010), Issue 10, pp686-689

[19] Lars Lydersen, Vadim Makarov and Johannes Skaar: Secure gated detection scheme for quantum cryptography, Physical Review A 83, 032306 (2011)

[20] W.K. Wootters, W.H. Zurek: A Single Quantum Cannot be Cloned, Nature 299 (1982), pp802-803

[21] G. Brassard C.H. Bennet and J.M. Robert. Privavy amplification by public discussion. SIAM Journal of Computation, Vol. 17 (1988), pp210-229

[22] N. Lütkenhaus: Security against individual attacks for realistic quantum key distribution, Physical Review A, Vol. 61, 052304 (2000)

[23] G. Brassard and L. Salvail. Secret key reconciliation by public discussion. Lecture Notes in Computer Science, Vol. 765 (1994)

[24] http://www.generalphotonics.com for the exact page(s) use the website's search option and search for PolaRite

[25] David Elkouss, Anthony Leverrier, Romain Alléaume and Joseph J. Boutros: Efficient reconciliation protocol for discrete-variable quantum key distribution, arXiv:0901.2140v1

[26] C.H. Bennett, G. Brassard, N.D. Mermin: Quantum cryptography without Bell's theorem, Phys. Rev. Lett. 68, 557 (1992)

[27] Bernhard Schrenk. Polarisationsnachregelung über lange Glasfaserstrecken fr Quantenkryptographie. Master's thesis, TU Wien, 2007.

[28] Thorben Kelling. Towards a HandsOff Plug & Play Quantum Key Distribution System with Entangled Photons for a Quantum Network. Master's thesis, University of Heidelberg, 2008.

[29] Daniele Ferrini. Active polarization stabilization of entanglement based QKD for deployment in a city fiber link. Master's thesis, University of Roma, 2008.

A. Curriculum Vitae

Andreas Allacher

Birthday	7th February 1986
E-Mail	andreas.allacher@univie.ac.at

Academic career

since 2005	Studies in physics, University of Vienna, matriculation number: 0501793
2000 - 2005	HTBLVA for EDV (IT) & Organisation Spengergasse (HTL) Spengergasse 20, 1050 Wien
1997 - 2000	Secondary school
1992 - 1996	Primary school

Vocational career

since 2012	Software developer, Cyberhouse
2002 - 2004	various internships during HTL

B. Zusammenfassung

Diese Arbeit repräsentiert ein Quantenkryptografie-System (auch QKD-System genannt), welches auf Verschränkung basiert für Glasfaserkabel des Telekommunikationsnetzes.
Es verwendet zwei Einheiten Alice und Bob und einen QRNG (Quantum Zufallszahlengenerator). Dieser QRNG wird zur Anbindung an das Tokio QKD Netzwerk benötigt und für die “Privacy Amplification” des Systems.

Die Alice Einheit beinhaltet eine Quelle für verschränkte Photonen und misst die 810nm Photonen dieser Quelle.
Die 1550nm Photonen derselben Quelle wandern durch ein Glasfaserkabel eines Telekommunikationsnetzes in die Bob Einheit, wo diese dann gemessen werden.

Das System läuft stabil über einen längeren Zeitraum durch die Verwendung von diversen automatisierten Stabilisations- und Optimierungsprozeduren.

Dem System ist es auch möglich einige der neu gefundenen Side-Channel-Angriffe (gefunden von der Gruppe von Vadim Makarov) zu erknennen und zu verhindern.

Des Weiteren wurde das System in das Tokio QKD Netzwerk zur Demonstration während der UQCC 2010 mit anderen QKD-Systemen integriert und
erreichte dort eine stabile Schlüsselrate und QBER.

Durch die Verwendung des Systems während der Demonstration im Tokio QKD Netzwerk hat sich herausgestellt, dass es möglich ist stabile Quantenkryptografie innerhalb von Städten (auch mittels verschränkten Photonen) zu betreiben.

Allerdings lässt sich auch noch einiges in Richtung Quantenkryptografie erforschen, vor allem in Richtung Side-Channel-Angriffe und natürlich in Richtung höherer Schlüsselraten.