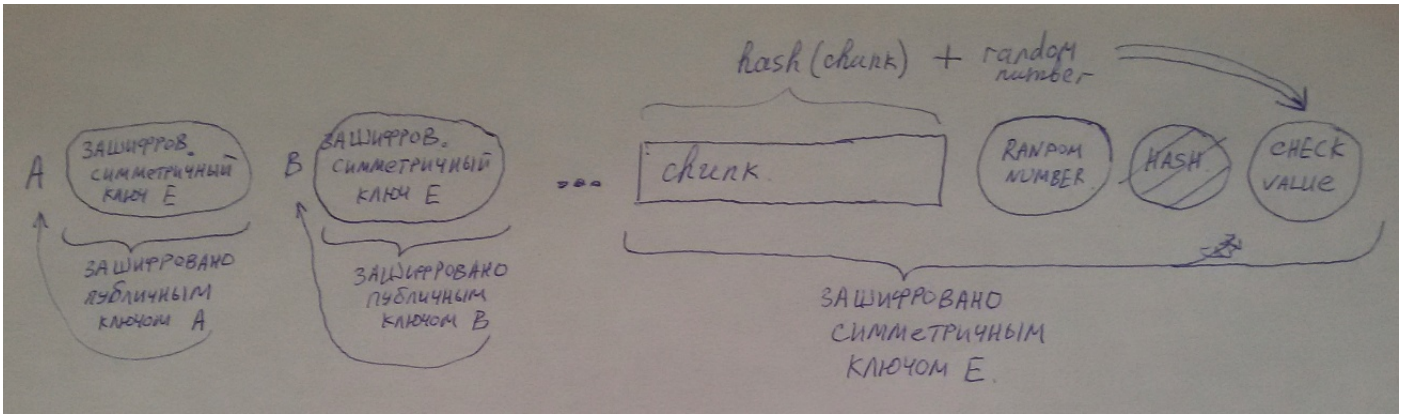
Схема формата «пакета данных»:



1) A, B – **публичные** ключи узлов A, B (они никак не шифруются).
Если же все (узлы, принадлежащие одному chunky) обменялись публичными ключами, то можно в A, B – хранить какие-нибудь короткие идентификаторы, т. е.какие-то числа, по которым однозначно можно определить публичный ключ соответствующего узла.

2) Вообще, основную часть пакета данных (т. е. Сам chunk) будем шифровать **симметричным** ключом **E** (*если допустить обратное, то возникает куча проблем, например, как рассылать широковещательное сообщение всем пользователям этого chunk... Или, учитывая, что, скорее всего, chunk – большая порция данных, нелогично использовать асимметричный подход, т. к. он проигрывает на 3 порядка симметричному*). Соответственно, мы как-то должны об этом ключе E сообщить другим пользователям этого chunk. Очевидно, что в открытом доступе нельзя его отправлять. Поэтому мы для каждого пользователя этого chunka зашифруем ключ E при помощи публичного ключа каждого пользователя. Следовательно, прочитать этот ключ E смогут только те пользователи, которые принадлежат этому chunky (*предполагаем, что узлы A, B держат в секрете свои приватные ключи!*).

Значит, если такой пакет данных приходит, например, к узлу B, то он начинает читать это сообщение, находит байты, соответствующие его публичному ключу. Далее, он своим приватным ключом дешифрует значение симметричного ключа E. И теперь при помощи него этот узел уже сможет посмотреть содержимое этого chunka.

Если же это сообщение получает злоумышленник (или узел, у которого нет доступа к этому chunky), то он не сможет узнать симметричный ключ E, при помощи которого и зашифрована основная часть – сам chunk данных.

3) далее идет сам chunk.

4) **Random number** и **check value.**
Зачем нужны random number и check value от него?
- Для проверки корректности.
Если узел B получает сообщение, далее читает ключ E. Читает данные chunk, вычисляет hash, и прибавляет random number. И сравнивает полученное значение с тем, что в пакете данных (check value). Если совпадают, то можно надеяться, что данные не повреждены и не заменены злоумышленником.

Random number получаем при помощи класса CryptoRandomGenerator, а для вычисления хэша – HashCoder.

Таким образом, можно предложить такой интерфейс для требуемых функций:

ChunkAddUser(bytevector const &oldChunk, bytevector const &pubKey):
В начало старого пакета данных oldChunk нужно добавить информацию о новом пользователе для этого chunka, т. е. Добавляем pubKey, и шифруем этим публичным ключом значение симметричного ключа E. Ясно, что пользователь, который не принадлежит этому chunky, не может добавить других пользователей,т.к. он не сможет прочитать значение ключа E из этого chunka.

ChunkCreate(bytevector const &chunkBody)

1. Сгенерируем симметричный ключ E для этого chunka (класс SymmetricalCoder)
2. Добавляем себя как пользователя, т. е. добавляем свой публичный ключ, потом при помощи него шифруем значение E.
3. Добавляем chunk (chunkBody).
4. Сгенерируем random number (CryptoRandomGenerator).
5. Вычислим хэш от chunkBody, складываем с random number и записываем результат в check value.
(шаги 3-5 – шифруются ключом E)

ChunkChange(bytevector const &oldChunk, bytevector const &newBody);

1. Вычисляем hash от newBody.
2. Сгенерируем random number. Складываем его с hashом на шаге 1, получаем check value.
3. Записываем зашифрованное ключом E: newBody, random number и check value.

Мы не меняем информацию о пользователях, поэтому старые пользователи этого chunk до сих пор имеют доступ к этому (уже измененному) chunky.

*При помощи классов **Serializer**, **Deserializer** мы упаковываем и распаковываем данные (например, можно пользоваться типизированным двоичным вариантом сериализации).*