Appendix B: Selected sample applications

1. *Levenshtein* Edit-Distance Calculator
   1. Description of Existing Application

This system allows user to enter two word into client console, which then requests a server to compute the *Levenshtein Distance*, LD, between the two words, where LD is the minimum number of single-character edits (insertion, deletion, substitution) required to change one word into the other. For example, the LD between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

* **k**itten → **s**itten (substitution of "s" for "k")
* sitt**e**n → sitt**i**n (substitution of "i" for "e")
* sittin → sittin**g** (insertion of "g" at the end)
  1. Current Design

Figure 1 shows an overview of the current architecture for this system. It only contains three classes, Client, calculator, and Message. Both the Client and calculator run as separate processes, and maybe even on separate machines. The Client allows the users to type in two words using a simple console interface. Then, it creates an instance of Message class containing these two words and sends it to the calculator. The calculator computes the LD, package that result up in a new instance of Message, and send it back to Client. The UML Sequence Diagram in Figure 2 shows this interaction.

Note, that the interaction is asynchronous from the Client’s perspective. In other words, the Client does not block while waiting for a response to the translation request.

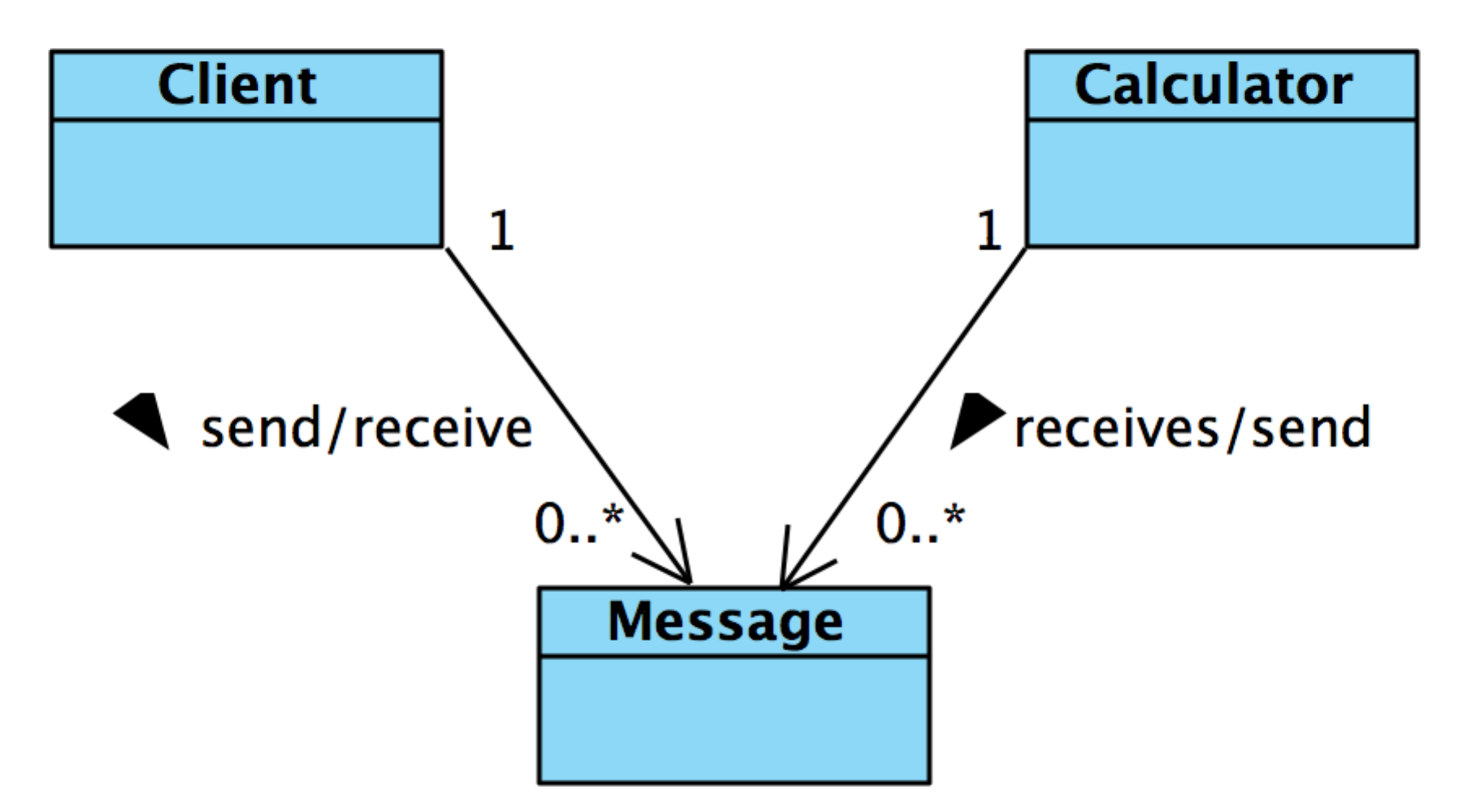


Figure 1: Architecture diagram of Levenshtein Edit-Distance Calculator

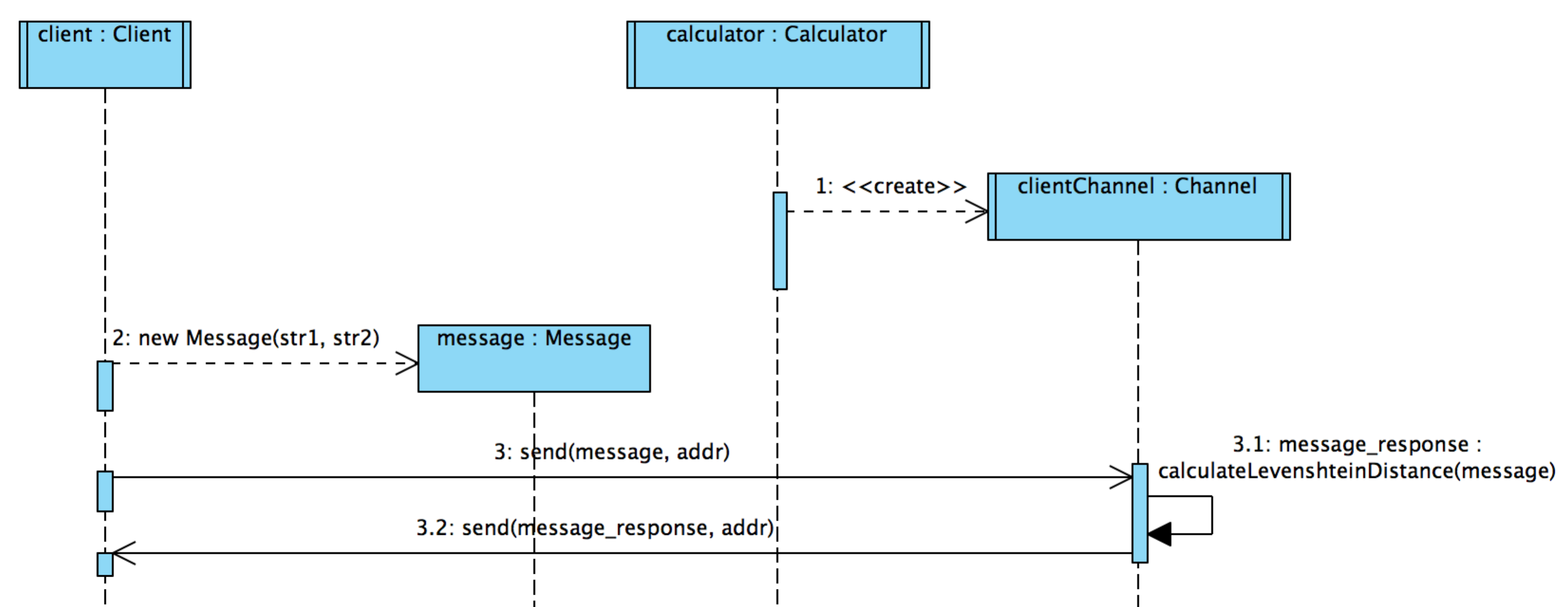


Figure 2: Interaction diagram between Client and Edit-Distance Calculator

1. *Symmetric-Key* Encryption
   1. Introduction

It encrypts the communication between a sender and receiver using symmetric-key encryption.

* 1. Description
     1. Exchanging secret keys
* The program first starts a KeyManager process, which handles the key requests from sender and receiver processes. We assume that both the sender and receiver are already registered with the KeyManager.
* Sender starts a KeyClient process, which sends a KeyRequest message to KeyManager. The KeyManager authenticates the sender, creates a SharedKey, encapsulates it in KeyResponse message, and sends it to sender.
* Receiver also creates a KeyClient, which sends a KeyRequest to KeyManager. The KeyManager again authenticates the receiver, creates a SharedKey, encapsulates it in KeyResponse message, and sends it to receiver.
* If KeyManager can’t authenticate any process, it sends an empty KeyResponse and the respective process terminates itself on receiving null Key.
* Figure (1 &2) describes the process of exchanging secret keys.
  + 1. Message Communications between Sender and Receiver
* Before sending a protocol message, Sender encrypts the message with the SharedKey.
* After receiving the message, Receiver decrypts the Message with the SharedKey.
  1. Application Material
* Developers would be provided with the following classes:
* *Encryption:* A data structure containing elements to measure performance.
* *KeyManager:* It authenticates the processes and provides the shared key.
* *KMClient:* It sends the authentication information to the KeyManager and requests the shared key.
* *KeyRequest:* A protocol message used to request the SharedKey.
* *KeyResponse:* A protocol message used by the KeyManager to send the SharedKey.
* *SharedKey:* This class encapsulates the shared key information.
  1. Figures

# 

Figure 1: Data Structures for Symmetric-Key Encryption

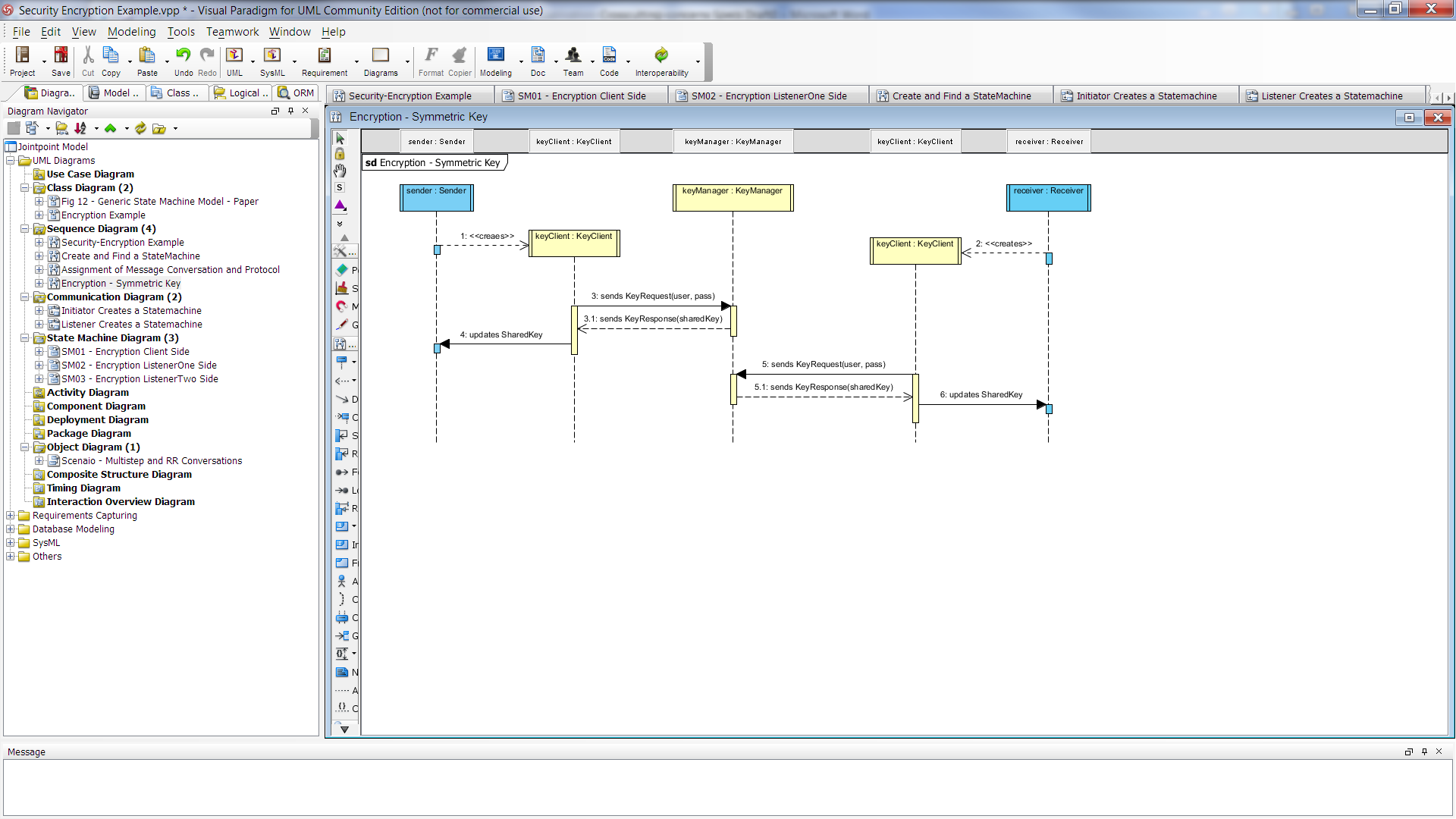


Figure 2: Process of exchanging shared keys