Based on forum post.

To implement an atomic counter, we can use atomic operations, more specifically the atomic_add operation.

The parameters to `atomic add` must be encoded in little endian two's compliment encoding. In Go, we can use "encoding/binary" to encode a 1 or -1 (in little endian two's compliment). Adding a negative 1 is then the same as a 1. In two's complement, -1 is entirely 0xff bytes, so you can hard code that in, or we could do something like this:

```
func incrKey(tor fdb.Transactor, k fdb.Key) error {
    _, e := tor.Transact(func (tr fdb.Transaction) (interface{}, error) {
        buf := new(bytes.Buffer)
        err := binary.Write(buf, binary.LittleEndian, int64(1))
        if err ≠ nil {
            return nil, err
        }
        one := buf.Bytes()
        tr.Add(k, one)
        return nil, nil
    })
    return e
}

func decrKey(tor fdb.Transactor, k fdb.Key) error {
    _, e := tor.Transact(func (tr fdb.Transaction) (interface{}, error) {
        buf := new(bytes.Buffer)
        err := binary.Write(buf, binary.LittleEndian, int64(-1))
        if err ≠ nil {
            return nil, err
        }
        negativeOne := buf.Bytes()
        tr.Add(k, negativeOne)
        return nil, nil
    })
    return e
}
```

Those two samples are very similar, but in the first, we increment the key by 1, and in the second we decrement by 1 (i.e., increment by -1). If we looked at the bytes, we would see that the first is a 1 byte followed by 7 0 bytes, and then the second is 8 0xff bytes.

Here's a sample that reads the given key, as well as a little tester than tries to read the key and do some atomic updates:

```
func getKey(tor fdb.Transactor, k fdb.Key) (int64, error) {
    val, e := tor.Transact(func (tr fdb.Transaction) (interface{}, error) {
```

```
        return tr.Get(k).Get()
    })
    if e ≠ nil {
        return 0, e
    }
    if val == nil {
        return 0, nil
    }
    byteVal := val.([]byte)
    var numVal int64
    readE := binary.Read(bytes.NewReader(byteVal), binary.LittleEndian, &numVal)
    if readE ≠ nil {
        return 0, readE
    } else {
        return numVal, nil
    }
}

func run(db fdb.Database) {
    var t tuple.Tuple
    t = []tuple.TupleElement{"foo"}
    var key fdb.Key
    key = t.Pack()
    db.Transact(func (tr fdb.Transaction) (interface{}, error) {
        tr.Clear(key)
        return nil, nil
    })
    v1, _ := getKey(db, key)
    fmt.Printf("v1 = %d\n", v1)
    incrKey(db, key)
    v2, _ := getKey(db, key)
    fmt.Printf("v2 = %d\n", v2)
    decrKey(db, key)
    v3, _ := getKey(db, key)
    fmt.Printf("v3 = %d\n", v3)
    incrKey(db, key)
    v4, _ := getKey(db, key)
    fmt.Printf("v1 = %d\n", v4)
    incrKey(db, key)
    v5, _ := getKey(db, key)
    fmt.Printf("v2 = %d\n", v5)
    decrKey(db, key)
}
```

If we run this, we should get:

```
v1 = 0
```

```
v2 = 1
v3 = 0
v1 = 1
v2 = 2
```

which is the desired behavior.