TOML: Project 1

Optimization of energy consumption and end to end delay in a wireless sensor network using duty-cycle MAC protocols

Arnau Abella
Universitat Politècnica de Catalunya

April 9, 2021

Listing 1 and 2 is reused through all three exercises. The α's and β's are only computed once per F_s since they are constant with respect to T_W.

```
def getAlphas(Fs) -> (float, float, float):
    d = 1 # where energy is maximized  (worst-case scenario)
    I = (2*d+1)/(2*d-1)
    Fi = Fs*((D**2 - d**2)/(2*d - 1))
    Fout = Fi + Fs
    Fb = (C - I)*Fout
    alpha1 = Tcs + Tal + (3/2)*Tps*((Tps + Tal)/2 + Tack + Tdata)*Fb
    alpha2 = Fout/2
    alpha3 = ((Tps+Tal)/2 + Tcs + Tal + Tack + Tdata)*Fout + ((3/2)*Tps +
    ↪  Tack + Tdata)*Fi + (3/4)*Tps*Fb
    return (alpha1, alpha2, alpha3)

def computeEnergy(Fs):
    alpha1, alpha2, alpha3 = getAlphas(Fs)
    def go(Tw):
        return alpha1/Tw + alpha2*Tw + alpha3
    return go

def computeDelay(Tw, Fs):
    d = D # where delay is maximized (worst-case scenario)
    beta1 = 0.5*d
    beta2 = (Tcw/2 + Tdata)*d
    return beta1*Tw + beta2

def exercise1():
    Fss = list(map(lambda x: 1.0/(x*60*1000), [1.0, 5.0, 10.0, 15.0, 20.0,
    ↪  25.0, 30.0]))
```

Listing 1: Functions for E^{X-MAC} and L^{X-MAC}

```python
def bottleneckConstraint(Fs, Tw: Variable):
    # Since the difference between ceiling and not ceiling is minimal,
    ↪  wlog we remove the ceiling.
    Ttx = (Tw/2)+Tack+Tdata
    I = C # If d=0 then I_d = C
    Fout1 = Fs*(D**2) # Fs*((D**2 - d**2 + 2*d - 1)/(2*d - 1)) where d =
    ↪  1
    Etx1 = (Tcs + Tal + Ttx)*Fout1
    return I*Etx1 <= 1/4
```

Listing 2: Bottleneck constraint

Exercise 1

Figure 1 is produced by listing 3. Each row in the figure corresponds to different values of F_s where $F_s \in \{5, 10, 15, 20, 25, 30\}$. The first column corresponds to the energy consumption (E^{X-MAC}) in *joules* of the *X-MAC* protocol with respect to the wake-up period (T_w) in milliseconds. The second column corresponds to the delay (L^{X-MAC}) in milliseconds of the X-MAC protocol with respect to T_w in milliseconds. The third column is just the combination of both previous columns in the same plot. The fourth column corresponds to the correlation between energy consumption and delay. From the last plot, we can conclude that minimizing both energy consumption and delay $w.r.t$ T_w is not feasible since there exist a negative correlation between these parameters. Notice, smaller values of T_w are not studied since those values are mathematically possible but not physically feasible.

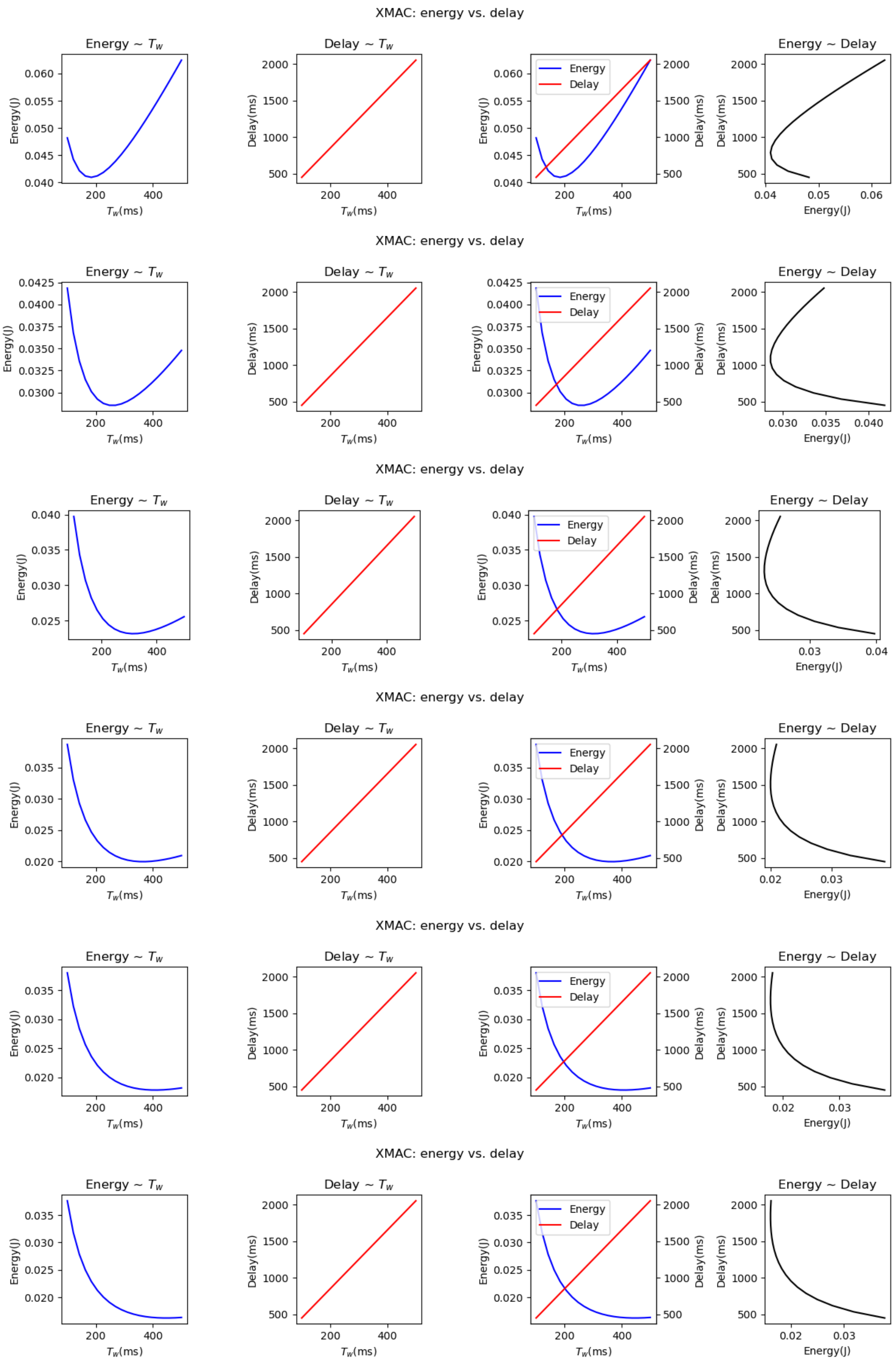

Figure 1: Energy vs. delay in XMAC protocol

```python
def exercise1():
    Fss = list(map(lambda x: 1.0/(x*60*1000), [1.0, 5.0, 10.0, 15.0, 20.0,
    ↪  25.0, 30.0]))
    Tws = list(np.linspace(Tw_min, Tw_max, num=20))
    arr = np.zeros((len(Fss),(len(Tws)), 2), dtype=float)
    for i, Fs in enumerate(Fss):
        for j, Tw in enumerate(Tws):
            arr[i,j, 0] = computeEnergy(Fs)(Tw)
            arr[i,j, 1] = computeDelay(Tw, Fs)
    # Plotting
    for i, subArr in enumerate(arr):
        fig, axs = plt.subplots(1, 4, figsize=(12, 3))
        # Fig 1
        axs[0].plot(Tws, subArr[:, 0], color='blue')
        axs[0].set_xlabel('$T_w$(ms)')
        axs[0].set_ylabel('Energy(J)')
        axs[0].set_title('Energy ~ $T_w$')
        # Fig 2
        axs[1].plot(Tws, subArr[:, 1], color='red')
        axs[1].set_xlabel('$T_w$(ms)')
        axs[1].set_ylabel('Delay(ms)')
        axs[1].set_title('Delay ~ $T_w$')
        # Fig 3
        axs[2].set_xlabel('$T_w$(ms)')
        axs[2].set_ylabel('Energy(J)')
        l1, = axs[2].plot(Tws, subArr[:, 0], color='blue', label='Energy')
        axcopy = axs[2].twinx()
        axcopy.set_ylabel('Delay(ms)')
        l2, = axcopy.plot(Tws, subArr[:, 1], color='red', label='Delay')
        plt.legend((l1, l2), (l1.get_label(), l2.get_label()), loc='upper
        ↪  left')
        # Fig 4
        axs[3].plot(subArr[:, 0], subArr[:, 1], color='black')
        axs[3].set_xlabel('Energy(J)')
        axs[3].set_ylabel('Delay(ms)')
        axs[3].set_title('Energy ~ Delay')
        # Plot
        fig.suptitle('XMAC: energy vs. delay', fontsize=12)
        fig.tight_layout()
        fp = plotsDir + 'exercise_1_{}.png'.format(str(i))
        fig.savefig(fp)
        print(f'(Exercise 1) {fp} written succesfully.')
```

Listing 3: First exercise

Exercise 2

The left plot of fig. 2 corresponds to the minimization of energy consumption subject to the maximum delay (L_{max}). This plot is computed by incrementing L_{max} in the interval $[500, 3000]$. T_w increases as the L_{max} increases (relaxes). Once the sweet spot is hit, relaxing L_w doesn't increase T_w.

The right plot of fig. 2 corresponds to the minimization of delay subject to the budget power consumption (E_{budget}). This plot is computed by incrementing E_{budget} in the interval $[0.1, 3]$. The plot is constant since E_{budget} is not restrictive (loose) and the maximum delay is achieved when T_w is minimum i.e. $L^{X-MAC} = L_{min}$.

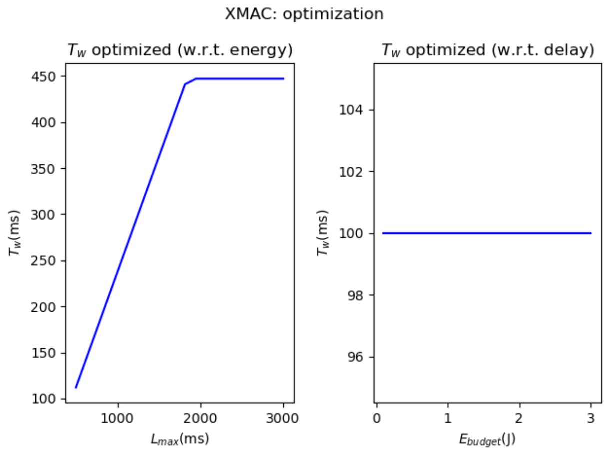

Figure 2: Optimizing XMAC protocol

The code for finding T_w that minimizes the energy consumption can be found at 4 and the code for finding T_w that minimized the delay can be found at 5. The problem can be solved in few lines of code thanks to the abstraction of functions 1 and mostly thanks to the `gpkit` library which does an incredible job for solving *geometric optimization problems.*

```
def p1(Fs, Lmax) -> float:
    """
    minimize E

    subject to:
      L  <= L_{max}
      T_w >= T_w^{min}
      |I^O|*E_{tx}^1 <= 1/4

    var. Tw
    """
    Tw = Variable('Tw')
    objective = computeEnergy(Fs)(Tw)
    constraints = [ computeDelay(Tw, Fs) <= Lmax
                  , Tw >= Tw_min
                  , bottleneckConstraint(Fs, Tw)
                  ]
    m = Model(objective, constraints)
    # m.debug() # Some problems are not feasible
    try:
        sol = m.solve(verbosity=0)
        return round(sol['variables'][Tw], 1)
    except Exception:
        return None
```

Listing 4: Minimization of energy subject to delay

```python
def p2(Fs, Ebudget) -> float:
    """
    minimize L

    subject to:
      E <= E_{budget}
      T_w >= T_w^{min}
      |I^0|*E_{tx}^1 <= 1/4

    var. Tw
    """
    Tw = Variable('Tw')
    objective = computeDelay(Tw, Fs)
    constraints = [ computeEnergy(Fs)(Tw) <= Ebudget
                  , Tw >= Tw_min
                  , bottleneckConstraint(Fs, Tw)
                  ]
    m = Model(objective, constraints)
    sol = m.solve(verbosity=0)
    # m.debug() # Some problems are not feasible
    try:
        sol = m.solve(verbosity=0)
        # Rouding to avoid numerical problems
        return round(sol['variables'][Tw], 1)
    except Exception:
        return None
```

Listing 5: Minimization of delay subject to energy

Exercise 3

By applying Nash Bargaining Scheme (NBS) we can solve the optimization problem which finds T_w that balances both E^{X-MAC} and L^{X-MAC}. The problem is non-convex although we can apply *wlog* the logarithm to the objective function to make it convex. The wake-up period that balances both energy consumption and delay is $T_w = 214.62$ milliseconds. Listing 6 solves NBS problem using `scipy` library.

```
def exercise3():
    Fs = 1.0/(30.0*60.0*1000.0) # arbitrary
    # Constants
    Eworst = computeEnergy(Fs)(Tw_min)
    Lworst = computeDelay(Tw_max, Fs)
    # Variables:
    # x[0] = Tw
    # x[1] = E_1
    # x[2] = L_1
    def objective(x):
        E_1 = x[1]
        L_1 = x[2]
        return - np.log(Eworst - E_1) - np.log(Lworst - L_1)
    def constraints(x):
        Tw  = x[0]
        E_1 = x[1]
        L_1 = x[2]
        E = computeEnergy(Fs)(Tw)
        L = computeDelay(Tw, Fs)
        return [ Eworst - E
               , E_1 - E
               , Lworst - L
               , L_1 - L
               , Tw - Tw_min
               , bottleneckConstraint(Fs, Tw)
               ]
    x0 = np.array([300.0, 0.02, 1000.0])
    ineq_cons = { 'type' : 'ineq',
                  'fun': constraints}
    res = minimize(objective, x0, method='SLSQP', constraints=[ineq_cons],
    ↪   options={'ftol': 1e-9, 'disp': False})
    p = round(res['x'][0], 2)
    print(f'(Exercise 3) Tw* = {p} milliseconds w.r.t. energy/delay')
```

Listing 6: Third exercise